On the Integration of Digital Signal Processors into Microkernel-based Operating Systems

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inform. Robert Baumgartl
geboren am 6. September 1970 in Mittweida

Gutachter:

Prof. Dr. rer. nat. Hermann Härtig, TU Dresden
Prof. Dr.-Ing. Heinrich Theodor Vierhaus, TU Cottbus
Prof. Dr.-Ing. habil. Dr. rer. nat. Winfried Kalfa, TU Chemnitz

Tag der Verteidigung: 20. 11. 2000

Dresden, im August 2001

Contents

List of Tables

List of Figures

1. Introduction

1.1. Motivation

In comparison to conventional personal computer systems, multimedia systems raise the following challenges for system designers:

- Guarantees of service qualities, once given, must be sustained.
- System latency must be kept low while data throughput must be kept high.
- The system must adapt to newly-arising algorithms and applications.

A system's designer is forced to choose a compromise between these requirements based on the main application area of the system. A high system latency is tolerated to a higher degree on a video-on-demand server than on a workstation. Relevant system parameters are allowed only to degrade gracefully, if at all, while multimedia data processing is performed. Of course, Multimedia processing places stringent demands onto hardware performance and operating system functionality.

A common approach to meet the challenges is to throw resources at the problem: use the highest-clocked CPUs, the fastest storage systems, a very large amount of memory and construct a high-performance system at high expense. A very defensive strategy is to tolerate service degradation up to a certain level by accepting that certain functions cannot be performed in parallel and tolerating transient system overload. This strategy is chosen in cost-sensitive areas.

A more promising approach is the strict control of resources like buses, caches or devices by privileged processes in combination with a microkernel-based operating system. Real-time processes may interact with the resource controllers to obtain *guaranteed* shares of scarce resources. Processes which are not able to provide service guarantees (e.g., time-sharing operating systems) are allowed to consume only resources which are not needed by real-time activities. In this way, conventional operating systems, as well as real-time components can coexist within a system. Of course, this carefully-crafted system structure does not improve the processing performance of the underlying hardware which is still critical for multimedia processing.

Therefore, this work focuses on all aspects of the integration of a so-called *accelerator subsystem* into such a system. By integrating inexpensive additional hardware into the system, we provide flexible, yet powerful means to relieve the system's CPU from computationally-intensive work and to reduce the CPU's interrupt load. Its heart is a processor which is, as we prove, ideally suited for multimedia data processing, a Digital Signal Processor (DSP).

DSPs are very common as stand-alone processors in consumer electronics, automotive and industrial applications requiring real-time services and high processor performance (for instance avionics or image processing). Without much success, it has been tried to integrate them into personal computer systems executing conventional operating systems (some of the results are portrayed in chapter 2). But up to now, no effort has been spent to integrate DSPs into microkernel-based operating systems as flexible, powerful accelerators maintaining real-time guarantees to client processes of the host.

The accelerator subsystem has to be tightly integrated into the host's operating system. To accomodate to actual operating system research, we mainly focus on microkernels here. We show how a microkernel executed at the host must be augmented to manage the subsystem accordingly. We compare several possible solutions, as *transparent threads*, a single server process and the combination of server process and specialized accelerator kernel and prove, why the latter approach is the most promising.

Very important for an optimal solution proved a careful decision process which part of the accelerator management has to be located within the host and which part should be located within the accelerator kernel. The work motivates the distribution of management services among CPU and DSP with respect to their different processor architecture. We provide insights, which kind of problems are ideal to execute with DSPs and which are not.

Components as accelerator workload estimation, admission control or scheduler are located at the host. We demonstrate why a form of *static scheduling* is optimal in terms of performance yet preserves a sufficient amount of flexibility. The scheduling algorithm is discussed in detail. Another interesting module is the workload estimation for DSP applications and for the DSP kernel itself. We demonstrate approaches to accurately estimate DSP code execution time and discuss the differences to conventional processors. Furthermore, we demonstrate how workload estimation is performed at system's runtime.

The main part of the work is dedicated to the design of a kernel specialized for accelerator subsystems. Step by step its structure and its services are derived. Design goals were efficiency and flexibility solely. For example, an optimum context switch implementation is described as well as simple yet efficient DSP process management services.

Special attention was paid to communication aspects between host CPU and DSP. We demonstrate the necessity of efficient communication mechanisms and argue that earlier projects failed mainly due to bad communication performance. We show that for different types of communication (e.g., stream data transfer, operating system signaling) different transfer mechanisms are essential. Furthermore, we believe that the kernel has to be implemented for every single accelerator processor, therefore porting aspects are discussed.

Moreover, we concentrate on subsystem applications. Several multimedia applications were implemented for the subsystem. The following performance analysis delivered valuable insight into accelerator optimization and helped to fine-tune the system. Some of the results are summarized in this work. We give detailed guidelines for the implementation of additional applications and derive general rules for optimizing DSP code for different processors. Last but not least, we pay attention to the partitioning of a given problem into parts to be executed by the accelerator and parts to be executed by the host processor.

1.2. Acknowledgements

This work would not have been possible without the support of numerous people. First of all I would like to thank my supervisor Hermann Härtig for his support and patience. He spent a considerable amount of his time discussing aspects of the work, contributing ideas and building up my confidence. During the later stages, Professor Härtig helped tremendously by carefully reviewing the text of the thesis.

Many helpful comments and constructive criticism came from the external advisors Theodor Vierhaus and Winfried Kalfa.

I would like to thank all my former teammates of Hermann Härtig's operating systems research group at Dresden University for their contributions, both technical and non-technical, especially Jean Wolter, Michael Hohmuth and Martin Borriss.

All members of the Graduate College “Tools for the Effective Use of Parallel and Distributed Computing Systems” deserve thanks for providing a very constructive research environment. The joint meetings have always been an ideal forum for extended discussions and learning.

Many thanks to my students Kai Stuhlemmer and Heinrich Langos for doing much implementation work and realizing some of the described ideas.

Special mentioning deserve my colleagues at Chemnitz University Frank Schubert, Frank Schöniger, Dietrich Thie and Sven Köhler for providing the necessary freedom to finish the thesis and for pushing me when it seemed necessary.

Mario Schubert did an invaluable job by identifying typos, tracking forgotten or unnecessary commas and improving the overall style of the text.

I'd like to express my gratitude to Gert and Brigitte Baumgartl for being such wonderful parents. I would not be where I am without them.

Last but not least, I am indebted to a number of friends for their help and their confidence during the course of this work, among them: Katharina Pietsch, Katrin Flemig, Dietmar Scharf, René Stange, Sven Rudolph and Heiko Schlittermann.

1.3. Layout

The remainder of this work is organized as follows:

Chapter 2 provides an overview over the state of the art in accelerator hardware components. Related projects are critically evaluated and their influence onto the project is characterized.

Chapter 3 concentrates on design aspects of the system. In the first part, the global system structure is derived from a requirement definition. The concept of a kernel and a dedicated host server is introduced and motivated. The second part of the chapter details the system. First, relevant host components are discussed and designed. Following, the structure of the accelerator kernel to be constructed is described. Fundamental design decisions are discussed and reasoned.

Chapter 4 is concerned with practical realization aspects. Some of the relevant implementation details for the accelerator kernel and the host components are presented. Additionally, the necessary structure of accelerator applications is described.

Chapter 5 proves the feasibility of the concept by providing and evaluating selected measurement results. We present obtained performance for raw DSP processing power, for DSP applications and for communication between host and accelerator subsystem. Additionally, criteria for the selection of accelerate-able applications are presented.

Chapter 6 summarizes the feasibility of the presented concept and main results of the work and gives an outlook of potential improvements and fields of further research.

2. The State of the Art

2.1. Processors

The following sections briefly discuss various approaches to microprocessor architecture to adapt to increasing demands of multimedia data processing. The subsections are ordered in increasing hardware specialization starting with instruction set extensions of General Purpose Processors (GPP), covering Microcontrollers and freely programmable Digital Signal Processors, continuing with Media Processors and finally closing with dedicated special processors and circuits.

2.1.1. Instruction Set Extensions for GPPs

Starting in 1996, several microprocessor vendors (among them Intel [Int97d], MIPS [Kil96] and HP [LH96], cf. table 2.1) introduced significant extensions of the instruction set architecture (ISA) to their established processor architectures. These extensions show the following similarities:

- targeted at multimedia applications (3D graphics, video, audio),
- Single Instruction Multiple Data (SIMD) architecture
- 64 bits data path width
- fixed-point or floating-point number format
- operand vectors with subwords length of 8, 16 or 32 bits
- one-clock-cycle-instructions mostly

The complexity of the new machine vector processing instructions differs. Every architecture implements elementary operations, such as transfer, addition, comparison and logical functions. Converting to and from the subword format ("packing", "unpacking") are also common. On the other hand, complex machine instructions as multiply-and-accumulate or averaging of several operands are also provided. The following two section briefly discuss the main aspects of the instruction set extensions. Additional material can be found in [SO97].

Vendor	ISA Extension	supported Processor Types
AMD	MMX	K6
	3DNow!	K6-2, K6-III, Athlon
Cyrix	Extended MMX	M2
HP	MAX	PA-8000, PA-8500 ...
IDT	MMX	WinChip C6
	3DNow!	WinChip 2
Intel	MMX	Pentium MMX, Pentium II
	ISSE	Pentium III
MIPS Technologies	MDMX	-
Motorola	AltiVec	PowerPC
Sun Microelectronics	VIS	UltraSPARC

Table 2.1.: Multimedia ISA Extensions for common Microprocessors

2.1.1.1. ISA Extensions of x86-compatible CPUs

The original MMX implementation introduced by Intel in 1996 [Int97d, Int97b, Int97c] comprises 8 new registers and 57 new machine instructions. The MMX registers are mapped onto the floating-point register stack preventing the parallel execution of MMX and floating-point code. The instructions operate on packed bytes (8 bits x 8), packed words (16 bits x 4), packed doublewords (32 bits x 2) or quadwords (64 bits). For compatibility to existing operating system software no new exceptions and state flags were defined. The highly unorthogonal MMX instruction set supports the following operations:

- add, subtract
- shift
- multiply and multiply-accumulate
- pack, unpack (conversion between different subword lengths)
- logical operations
- compare
- transfer

A special instruction leaves the MMX registers in a well-defined state to allow floating-point instructions to execute within MMX code. If overflow occurs due to addition or multiplication, the bits outside the subword range are either lost ("wrap-around") or the result is the extremum of the corresponding value range (saturation).

The implementation of MMX in AMD's processor AMD K6 is fully compatible to Intel's [Adv97]. In June 1998 AMD introduced an extension to MMX, "3DNow!", with the processor AMD K6-2 [Adv98]. It provides an additional number format which packs two 32-bits-wide floating-point numbers into a 64-bits-wide MMX operand and 21 new machine instructions implementing the typical MMX operations with the new data type. Complex operations as reciprocals or square roots are divided into several machine instructions. An approximation of the desired result is generated by a certain instruction which can be accompanied by further instructions to enhance the precision of the result, if necessary. Furthermore, 3DNow! offers the following new instructions:

- `PAVGUSB`, for summing up and averaging 8 bytes (intended for motion estimation and video scaling),
- `PREFETCH`, for fetching a complete line into the first level cache,
- `FEMMS`, a faster version of the original `EMMS` instruction (which brings all MMX register to a defined state).

A published performance evaluation claims a speedup of 6 in comparison to FPU code [Sti98]. With the processors K6-III and Athlon, AMD slightly improved 3DNow!: Both processors provide two 3DNow! units (in contrast to one of the K6-II) giving them the possibility to execute 2 MMX and 2 3DNow! instructions in parallel. Additionally, 2 new MMX/3DNow! instructions were added. First performance benchmarks reveal roughly equity between ISSE (described below) and 3DNow! of the K6-III [Sti99].

In late 1997, Integrated Device Technology (IDT), introduced another Pentium-compatible CPU [Cen97]. The MMX unit of the IDT WinChip C6 is fully compatible to Intel's MMX. IDT claims a substantial performance advantage without providing proof. Its successor, the WinChip 2, provides two MMX units including full 3DNow! support [Pos98].

Cyrix, introduced "Extended MMX" as ISA extension to its latest-generation microprocessors [Cyr97a, Cyr97b]. Extended MMX is a superset of Intel's MMX. The 12 additional machine instructions have to be enabled separately. The extension tries to overcome the disadvantage of original MMX that the result of an operation always overwrites one of the source operands, which is typical for a 2-address-machine. Instead, some of the machine instructions use an implicit third operand[1] as destination thus preserving both source operands. The additional instructions provide the following complex functions:

- multiply-and-accumulate with rounding
- accumulation of byte distances
- conditional moves
- determination of maximum magnitude
- averaging of 2 operands

[1] a different MMX register which depends entirely on the first source operand

The additional instructions are likely to speed up video algorithms, for example motion estimation and image filtering.

Halfhill [Hal97] states that in the near future x86 MMX will very probably diversify trying to accelerate single-precision floating-point calculations. Besides the already mentioned "3DNow!" extension for the next generation of AMD processors, IDT and Cyrix also announce extensions to their current MMX implementations. IDT prospects 30 new floating-point registers of 80 bits each as well as 24 new instructions including a multiply-accumulate operation. The next version of Cyrix's M2 processor incorporates "MMX-FP", which will enhance reciprocals, reciprocal square roots and motion-estimation using a packed floating-point number format and 12 new instructions.

The latest x86 architecture extension was Intel's *Internet Streaming SIMD Extension*[2] (ISSE) introduced with the Pentium III processor in February 1999 [KP99], [TH99]. In contrast to all processor extensions previously described, ISSE introduces 8 new registers of 128 bits length each, which can be accessed independently of CPU, FPU or 'classic' MMX registers. This concept eliminates a major bottleneck of MMX but gives up full compatibility to pre-ISSE processors.

The number format is identical to 3DNow!, therefore, ISSE operates onto four floating-point operands in parallel. It provides 72 new instructions, mostly to adapt the conventional MMX functionality to the new number format. The complex operations include shuffle instructions to change data layout in memory, averaging and the simultaneous computation of minimum and maximum of 4 operands. Furthermore, instructions for software-controlled cacheability are provided.

2.1.1.2. ISA Extensions of x86-incompatible CPUs

Outside the market of Personal Computers with x86-compatible CPU compatibility between CPUs of different vendors is no concern, so every microprocessor vendor implements its own concept of multimedia ISA extensions although similarities are obvious. The following section briefly introduces these these extensions.

One of the first processors to implement instructions with operands of packed subwords was the DEC Alpha AXP 21XXX [Dig96]. Its 13 "Multimedia Support Operations" use the processor's global register set. The instructions allow to determine minima and maxima in the subwords, to convert between different number formats and to compute the sum of absolute differences of the subwords. The latter instruction is destined to speed-up the macroblock search in motion estimation.

Sun's Visual Instruction Set (VIS) was introduced in 1995 [Sun95] and intended for graphics acceleration. It shares the processor's 32 general-purpose registers with floating-point code. Transfer operations may bypass all caches by directly accessing main memory thus increasing copy bandwidth. Besides elementary operations, VIS offers an instruction to compute the sum of absolute subword differences, support for traversing arrays and a merge command to efficiently separate color components of video data. It does not provide instructions for shifting and for multiplying and adding in parallel.

[2]Many publications refer to ISSE as "Katmai New Instructions" (KNI).

Hewlett-Packard introduced a first multimedia-dedicated ISA extension with their PA-7100LC and PA-7300LC processors. The new 64 bits wide chip architecture HP PA-RISC 2.0, starting with the PA-8000 processor, contains a revised and enhanced extension, called "Multimedia Acceleration eXtensions" (MAX) [LH96]. The 31 integer general-purpose registers may be used as any of the 3 operands. Like MMX, MAX supports signed and unsigned saturation. Besides elementary operations, like add/subtract, shift and transfer, operations for averaging and for subword rearrangement[3] are included.

MIPS extends its latest-generation microprocessor architecture in two ways. First, the new MIPS V instruction set supports a new number format called "paired-single", which packs two floating-point subwords of single precision into a 64 bit wide operand. Second, an MMX-like extension, MIPS Digital Media Extension (MDMX) has been introduced [Kil96, MIP97]. Like Sun's VIS, only 8 and 16 bits wide subwords can be processed in parallel and the 32 VIS registers are shared with floating-point code. A unique feature is the 192 bits wide accumulator which avoids arithmetic overflow in nearly all situations. The MDMX instruction set is very comprehensive. Instead of a sum-of-absolute-differences function an instruction which sums the squares of absolute differences has been defined providing an even better video motion estimation performance [MIP97]. Still, processors implementing the new features have to be introduced to market.

The last microprocessor family upgraded with a SIMD extension for multimedia processing is the PowerPC by Motorola, IBM and Apple. The extension named "AltiVec" [Ful98] was introduced in May 1998. Until then, all PowerPC vendors propagated third-party Media Processors (cf. section 2.1.4) as multimedia accelerators but never realized that concept. Among other, AltiVec features a 32x128 bits vector register file, 162 new machine instructions and saturation arithmetic. The unit is able to process 8, 16 and 32 bits wide integer vectors as well as single precision floating-point vectors. A unique feature is the permute operation[4], which is capable of arbitrarily selecting data with the granularity of a byte from two 16-byte registers and assembling it into a single 16-byte destination register. Motorola claims a performance better than state-of-the-art DSPs without publishing exact benchmark results.

It can be concluded that multimedia ISA extensions for workstation CPUs are more advanced in comparison to personal computer CPUs promising a better acceleration of the targeted algorithm but intentionally giving up compatibility to earlier processors of the same vendor.

2.1.2. Microcontrollers

Another class of microprocessors which can be taken into consideration are microcontrollers. In contrast to general-purpose microprocessors, microcontrollers are targeted at more cost-sensitive application areas, for instance embedded systems or consumer electronics. Especially, low power consumption is a must.

[3] This is useful for speeding-up matrix operations.
[4] This operation is useful for pixel manipulation, for instance zooming.

Frequently, microcontrollers are connected to a DSP (cf. section 2.1.3) which works as a slave processor for computationally-intensive tasks. The microcontroller supervises the DSP and is responsible for general-purpose tasks, for instance user interfacing. This heterogeneity led to the present trend of integrating architectural aspects of DSPs into the microcontroller architectures to provide a unified programming interface for both control-oriented and DSP code. Two main development directions can be distinguished:

Either, a complete microcontroller core and a complete DSP core are integrated onto one chip together with assorted memory and facilities for inter-processor communication. Both processors remain separate and are independently programmable. As an example, the Motorola MC 68356 includes a DSP 56002 core and a MC 68000 microprocessor core [Mot94].

Or, the microcontroller core is extended by certain DSP functional units and the instruction set is adapted accordingly. Prominent representatives of this category are the Piccolo extension of the ARM family of Microcontrollers [Wal96], [RW97] by Advanced RISC Machines or the Hyperstone E1 [DBJ+98].

Although the ARM Piccolo DSP extension to the family of ARM microcontroller cores is basically a processor extension, it is equipped with a separate program counter. The microcontroller core initializes the program counter and is then free to execute its own RISC code. However, it is responsible for data transfer between memory, microcontroller core and Piccolo. This sharing of a single bus limits the I/O bandwidth in comparison to a DSP's Harvard architecture. Among other the Piccolo features zero-overhead looping, single-cycle multiply-and-accumulate operations, saturation arithmetic and 48 bits wide accumulators. DSP benchmarks and a comprehensive description of the architecture can be found in [Wal96].

The Hyperstone E1-32XP [DBJ+98], [Sch96] has a similar concept. The 32-bit microcontroller core is extended with a 16x16 bit multiplier, an adder, a barrel shifter and an accumulator. Additionally, this processor features delayed branches[5], which avoid pipeline stalls due to branching.

Similar architectural developments can be observed in the Siemens TriCore family [Mar98] and in the Hitachi SH-DSP microcontroller [Hit98].

Obviously, microcontrollers are also suitable as accelerator processors. They were not selected in favor of DSPs for the following two reasons:

1. Contrary to the claims of microcontroller vendors, moderately-priced microcontrollers with DSP extensions do not reach the performance of present DSPs.

2. Microcontrollers include certain on-chip functional units which are important for their intended application areas (e.g., sophisticated serial interfaces) and which are useless for PC multimedia accelerators. This slightly deteriorates the price-performance ratio of microcontrollers in comparison to DSPs.

[5]Delayed branches are common for DSPs, e.g., the TMS320C4x.

2.1.3. General-Purpose Digital Signal Processors

This section briefly covers important aspects of general-purpose Digital Signal Processors (DSPs)[6]. The architectural variety among Digital Signal Processors is much broader than among general-purpose microprocessors [LBSL97]. Currently, there exist dozens of DSP families on the market [BLS+97]. Across different vendors, DSPs are not compatible. Several approaches have been made to categorize and systematize DSP architectures, with respect to performance, prize or architectural features. Figure 2.1 gives an overview with some examples.

As can be seen, the most important characteristic feature is the number format of the processor. Another distinction can be made using the data path width. Whereas floating-point DSPs always use a data path of 32 bits, fixed-point DSPs have either 16 or 24 bits, depending on the prospective area of usage. Furthermore, one can establish different performance classes for DSPs of all respective categories.

The vast majority of DSPs use fixed-point arithmetic and are applied in very cost-sensitive areas. In contrast, floating-point DSPs are more expensive, because floating-point ALUs require more die size than their fixed-point counterparts. On the other hand, floating-point code can be developed more rapidly, because the programmer does not need to pay attention to rounding errors, word length and scaling effects. Floating-point DSPs are frequently used as computing nodes in multiprocessor systems.

2.1.3.1. Characteristics of DSPs

Regardless of the diversity, some architectural features are common to all DSP types.

- The transfer path between processor and memory is highly optimized.

  Signal processing applies algorithms to streams of digitized signal data. The necessary high data throughput is guaranteed by a Harvard architecture. Most DSPs support 2 or 3 memory spaces which can be accessed simultaneously. Instead of dynamic RAM static RAM is used which drastically reduces access times and increases cost.

- The execution timing of DSP code is very accurately predictable.

  DSPs are very frequently used under real-time constraints. That's why, DSP architectures are designed to guarantee exact predictability of execution timing rather than to provide maximum performance. This is the reason for the main architectural differences to general-purpose processors. Especially, DSPs do not utilize hard-to-predict processor units and functions, like

  - complex cache hierarchies,
  - virtual memory,

[6] In the following, we use the term "general-purpose DSP" to emphasize the free programmability in contrast to "specialized DSPs" which are able to perform only a limited number of different DSP algorithms.

  - branch prediction,
  - speculative execution,
  - deep processor pipelines.

- DSPs provide very efficient implementations of multiplication and addition operations.

  Typically, DSPs provide a machine instruction which performs a multiplication and an accumulation (of the preceding product) in one clock cycle, often accompanied by a rounding operation, called "multiply-and-accumulate" (MAC) As an example, the ADSP-2106x SHARC by Analog Devices performs 2 floating-point MACs in one clock cycle. This MAC operation is the basis of many signal processing algorithms, for instance digital filters. A Finite Impulse Response (FIR) Filter is calculated as follows:

  $$y = \sum_{t=1}^{N} x(T-t) * c_t$$

  Multiplication and addition are also very common in multimedia-oriented algorithms.

- DSPs offer many micro-controller features.

  DSPs are often used in embedded systems. As a consequence, DSPs are usually equipped with on-chip peripheral units, as serial or parallel interfaces, RAM and ROM, timers, DMA controllers and debugging capabilities. DSPs do not need chipsets to interface to the outside world. It is possible, to run DSPs in stand-alone mode, booting and controlling them via an external interface.

- DSPs provide special addressing modes.

  Certain signal processing algorithms can considerably be accelerated when specialized addressing mechanisms are used. For instance, there exist a number of Fast Fourier Transform (FFT) algorithms which leave their respective results in a seemingly strange order [OS95]. To reorder the results, most DSPs provide the so-called bit-reversed addressing mode. Another example are addressing modes to implement circular buffers with no overhead.

As can be seen, DSPs could operate as dedicated co-processors. The following overview summarizes the main reasons for our suggestion of the integration of DSPs as separate multimedia accelerators into future personal computer architectures.

- DSPs are freely programmable and suitable for a broad range of applications.
- DSPs are cheap, they do not require external chipsets. Therefore, their price-performance-ratio is much better than that of general-purpose processors.
- DSPs are easy to interface to peripheral devices and other processors.

- DSPs are designed to handle large amounts of streamed data. The accelerator system to be constructed specializes in processing of streamed data.
- DSP are designed to operate under heavy interrupt load.
- DSPs have a well predictable execution timing. As will be shown in detail in chapter 3, especially section 3.3.1.3, accurate execution timing is essential for operating a DSP with a high load under real-time constraints.

For our experiments, the Texas Instruments TMS320C44, clocked at 50 MHz, was chosen. Its floating-point number format supported the rapid development of sample applications as described in chapter 4. A more in-depth discussion of the project's hardware can be found in the appendix, the processor is thoroughly described in [Tex96b] and [Tex96a].

2.1.3.2. Benchmarking DSPs

Considerable effort has been spent already in evaluating DSP performance. Due to the variability of DSP architecture, benchmarking DSPs is a complex task. There exists even a company, Berkeley Design Technology, Inc.[7], which specializes in benchmarking DSPs.

It is beyond the scope of this work, to review and discuss all relevant publications concerned with DSP performance evaluation. Instead, we present and discuss a representative selection.

When DSPs became freely-programmable (and, hence, were able to perform every possible task), the necessity of comparing DSPs to conventional processors arose. In 1992, Stewart et al [SPL92] raised the provocative question, whether DSPs are in general obsolete. The authors performed a small suite of benchmarks with the DSPs TMS320C40 and TMS320C30 as well as the GPP Alpha AXP21064 and concluded from the obtained results that DSPs would extinct in the near future, because GPPs would be able to perform signal processing faster than dedicated DSPs. The continuing existence of DSPs has disproved this allegation so far.

The Institute for Integrated Systems in Signal Processing of Aachen University of Technology developed the DSPstone benchmark methodology to compare C compiler efficiency for both DSPs and GPPs [ŽMSM94]. In [ŽSWS95] the authors apply the benchmark to multimedia-type applications. The result is somewhat contrary to the conclusion of Stewart et al: "It can be concluded that for DSP-type code DSP processors cannot be replaced by their powerful GPP relatives." ([ŽSWS95], pg. 5).

Many publications study the TMS320C6x DSP, one of the most powerful processors available. [Lou97] compares the C6x with the TMS320C4x[8], and with the Analog Devices SHARC, another high-end floating point DSP. Special attention is paid to the performance of the C compiler. As a main result the C6x outperforms the competing

[7] `http://www.bdti.com/`

[8] The TMS320C4x is used for the practical experiments of this project.

DSPs at least at a factor of five. The analyses in [ST97] and [CHE97b] reveal similar results.

In [Lap94] the author proposes DSPs as multimedia accelerators, provided the following exists:

- powerful, yet inexpensive DSPs,
- DSP application code,
- DSP support software (operating systems for DSPs, DSP 'Managers' for PCs),
- host applications exploiting the accelerator.

The paper's argumentation is based on DSP performance only. No details on the accelerators to be constructed or the host systems are given.

A whole suite of publications concerning DSP benchmarking has been released by Berkeley Design Technology, Inc. [Bla97, WB97, Bie97]. The authors develop the following benchmark methodology: A defined set of algorithm "kernels"[9] is implemented for a given processor using the highest possible degree of optimization. The suite consists of five different signal-processing filters, three vector operations (dot product, add, maximum estimation), a convolutional encoder, a finite state machine and one Fast Fourier Transform (FFT). Then, execution times are measured.

Finally, the obtained execution are combined into one aggregate number describing the performance of the evaluated DSP, the BDTImark.

2.1.4. Media Processors

The trend of providing multimedia services with PCs as well as with stand-alone devices has led to yet another class of microprocessors, the *Media Processor*. The first two representatives of media processors were the Mpact R chip by Chromatic Research, SGS-Thompson and Toshiba [Chr97] and Philips' TriMedia TM-1000 processor [GS96, Phi97]. Both processors feature a freely programmable VLIW core and are specialized to perform multimedia acceleration in at least four ways:

1. *Data format:* Typical multimedia data formats such as 8 or 16 bit video and audio and floating-point 3D graphics data are native data formats of media processors making conversion operations obsolete.

2. *Functional units:* To relieve the media processors core from computationally intensive tasks special units are integrated (e.g., an image co-processor for filtering and scaling or a variable length decoder for decoding Huffman-encoded media data [Phi97]). As an example, the block diagram of the TriMedia processor is depicted in figure 2.2.

[9] The term "kernel" emphasizes that only the core of an algorithm is actually implemented instead of a fully-featured application. It has nothing in common with an operating system kernel.

3. *Instructions:* Many frequently encountered multimedia operations are available as machine instructions (e.g. summing and averaging for motion estimation).

4. *Interfaces:* Several hardware media interfaces are integrated onto the chip, for example for digital video and audio (SP/DIF) streams, for musical data (MIDI) and for connection to a host PC (i.e, PCI).

Media processors resemble general-purpose DSPs very closely. However, in contrast to the latter, the vendors state that all software development for media processors should be done in HLL and an optimizing compiler is responsible for optimal instruction scheduling.

Media processors are intended to operate as PC coprocessors as well as in stand-alone devices (e.g., DVD players, multimedia television sets, etc.) at a substantial lower prize than other accelerator concepts. Possible drawbacks of the concept are:

- Media processors are targeted at the consumer market. As a consequence decoding algorithms are more reflected in instruction set and chip architecture than encoding algorithms.

- The specialization discussed above may lead to inflexibility with regard to newly developed algorithms and standards.

- Exact execution times are hard to predict, because of the inherent chip complexity and the necessity for compilers. This complicates the development of real-time applications for these processors.

At the moment acceleration hardware for PC architectures, low-cost graphics accelerators with 2D and 3D capabilities as well as DVD playback support, is introduced to the market [Ber98]. Nevertheless the majority of media processors is used in stand-alone devices [RSV97].

2.1.5. Special Processors

In contrast to Digital Signal Processors which are in principle able to execute every potential algorithm Special Processors are only able to execute a finite set of algorithms. The speed-up for the targeted algorithms in comparison to DSPs can be enormous. As an example, a typical representative of this type of microprocessor is discussed and evaluated.

2.1.5.1. TM-66 swiFFT DSP Chip

The swiFFT Chip [Mar96, MM96] introduced by Texas Memory Systems in 1996 is one of the most powerful FFT processors available today. It features a Harvard architecture, a total of 20 floating-point adders and multipliers, several register files and is clocked at 40 MHz. Its 2 pipelines of 3 stages each can be configured as one complex data pipeline or two real data pipelines and are programmable in a limited way. The stages perform the following operations either on real or on complex data:

- stage 0: addition, subtraction, multiplication, absolute value, conjugation
- stage 1: addition, subtraction
- stage 2: addition, subtraction, multiply by j, complex-conjugate

Four input buses and two output buses of 32 bit each guarantee an I/O bandwidth matching the processing power of the processor but require a high-speed interface to external memory.

The processor is intended primarily to execute the radix-16 butterfly FFT with real or complex operands. Due to its (limited) programmability it is also possible to perform similar elementary algorithms with a certain performance degradation as indicated by table 2.2.

Algorithm	Data Size	Time	MFLOPS
Complex FFT	1K	88s	680
	16 K	1.5 ms	680
	1M	132 ms	680
Real Matrix Multiply	1Kx1K	3.36 s	640
Complex Matrix Multiply	1Kx1K	13.4 s	640
Complex Vector Multiply	1K	25.6 s	240
Real Vector Add	1K	12.8 s	80

Table 2.2.: TM-66 swiFFT Performance at 40 MHz [Mar96]

The following elementary algorithms can be performed [MM96]: FFT, convolution, correlation, matrix multiplication, vector addition. The vendor of the processor develops large DSP systems composed of several swiFFT processor modules and claims typical real-time and non-real-time DSP tasks (e.g., radar and sonar processing, medical imaging, image compression) as primary application areas.

There exist a wide range of special circuits or chipsets for the execution of algorithms especially for multimedia processing which are either non-programmable or programmable only in a very limited way. For example, chipsets for DVD devices exist, which provide MPEG audio and Dolby AC-3 encoding or decoding functionality.The inherent inflexibility of these special processors prevents their usage within the context of this work.

2.1.6. Field Programmable Gate Arrays

Another integrated circuit potentially suitable as accelerator is the Field Programmable Gate Array (FPGA). These circuits consist of a large number of standard logic cells which can be configured by a controlling processor at any time (hence the term *reconfigurable computer* for such systems), and, in this way, can be adapted to different algorithmic requirements. At the moment, FPGAs are not complex enough to synthesize a fully-featured (co-)processor. Instead, only a well-defined subset of processor functionality is realized within the FPGA.

[Kna95] discusses several FPGA implementations of typical signal processing algorithms, compares them to DSP implementations and estimates the theoretically achievable speed-up. For more complex algorithms (i.e., a Viterbi Decoder) the author proposes a combination of a general-purpose DSP and a FPGA. The FPGA accelerates elementary operations with high data throughput whereas the DSP performs the complex parts of the algorithm and configures the FPGA.

Wittenberger and Härtig [WH95] discuss the application of an FPGA as flexible external device in standard PC architectures and propose several methods to integrate the new device into existing host operating system structures. They propagate microkernel-based OS architectures which can be adapted to flexible hardware more easily[10] in comparison to monolithic operating systems. Because of the comparatively low complexity of a stand-alone FPGA a connection to a DSP is proposed. As possible application areas the authors suggest multimedia codecs, (partial) network protocol stacks and image rendering.

Reconfigurable architectures use another integration approach. The aim of the project described in [Wei97] is the automatic extraction and generation of coprocessors from high-level language (HLL) program code. The author states that regular iterative computations which perform the same operation on a large set of data (i.e., loops) are most likely to benefit from FPGA acceleration. The described methodology identifies loops, simplifies them and synthesizes them as pipelines on a connected FPGA. A so-called *pipeline control unit* which is also located on the FPGA realizes the data transfer between host and FPGA.

The theoretical speedups for the given examples (image smoothing, FIR filter) are very promising (ranging from factor 20 to 40) but the necessary overhead introduced by communication and reconfiguration lowers it to $1.28 \ldots 1.9$. This emphasizes the importance of very efficient interfaces between accelerator and host CPU. Complex operations (e.g., floating-point calculations) are excluded from acceleration due to limitations of actual FPGA technology.

To summarize, the potential acceleration by an FPGA is very high in comparison to DSPs. On the other hand, the migrated functionality is of low complexity; only elementary functions can be executed by an FPGA. Additionally, the necessary reconfiguration requires more time than downloading new code to a DSP. Synthesizing new algorithms is more expensive than implementing it for a general-purpose DSP.

[10] No changes to the OS are necessary to adapt to different FPGA configurations. Instead, a user-level server consisting of two threads responsible for communication to the FPGA configures and controls the FPGA at system runtime.

2.2. Interfaces between CPU and Accelerator

The following sections briefly discuss the state of the art of peripheral interfaces for industry standard PCs with respect to their potential applicability as hardware connection mechanism between host and the accelerator hardware.

In this discussion, we solely focus on interfaces and protocols specialized for communication with peripheral devices. In the future, these interfaces could be substituted by networking protocols and interfaces as proposed in [MFH98]. The authors used IP and UDP via Myrinet to communicate with network-attached peripherals and claim to reach 80% of the maximum SCSI performance. It is possible to extend this approach to accelerator hardware, but this is beyond the scope of this work.

2.2.1. Local I/O Buses

This section examines *local I/O buses*, that are, buses which connect peripheral devices to the system's CPU and main memory. Buses which are obsolete by the time of writing are omitted although their performance would justify a closer examination. Therefore, the following local I/O bus architectures are not considered:

- VESA Local Bus (VLB),
- Extended Industry Standard Architecture (EISA),
- Micro Channel Architecture (MCA).

Instead, we concentrate on Industry Standard Architecture (ISA), Peripheral Component Interconnect (PCI) and Accelerated Graphics Port (AGP).

2.2.1.1. Industry Standard Architecture

Industry standard architecture (ISA) has been the quasi-standard for the industry standard PC local I/O bus for nearly 15 years[11]. Nevertheless, it provides reasonable throughput for devices with low bandwidth requirements. The basic technical characteristics are:

- width: 16 bits
- clock speed: 8.33 MHz
- maximum transfer rate: 8.3 MBytes/s

Intel plans to remove ISA from the future Intel Architecture-based PC platform (cf. the Visual Computing Initiative [Int97a]). Additionally, there are publications, which indicate, that ISA is not an ideal hardware interface for connecting CPU and accelerator subsystem [BH96], [Bau95].

[11] The latest modification to ISA was the transition from 8 bits to 16 bits in 1984.

2.2.1.2. Peripheral Component Interconnect

Peripheral Component Interconnect (PCI) is the fastest local I/O bus currently available for the PC hardware platform. It was developed by a company consortium led by Intel and was introduced in 1993 [SA95]. Later, PCI was adopted by other computer manufacturers, too. The actual specification 2.0 defines the following features:

- 32 bits wide
- 33 MHz maximum clock speed
- theoretical maximum transfer rate 133 MBytes/s[12]
- auto-configuring
- IRQ sharing
- bus mastering (data transfer between memory and PCI devices without CPU intervention) of multiple devices simultaneously
- Burst mode (after an initial address is provided, sets of consecutive data words may follow)

Its availability for a range of computer platforms, its standardization and its performance make PCI an excellent candidate as connection mechanism between host and accelerator subsystem.

2.2.1.3. Accelerated Graphics Port (AGP)

The Accelerated Graphics Port (AGP) [Car96], [Int98] is a superset of the PCI 2.1 specification. It was introduced by Intel in 1997. The intention of its development is the acceleration of three-dimensional graphics beyond the capabilities of the PCI bus (cf. section 2.2.1.2). Strictly spoken, the AGP is not a bus, because it is a point-to-point connection between the graphics controller and the main memory. Therefore, it cannot be a substitute for the PCI bus. Unlike PCI, AGP is intended only for the standard PC hardware platform.

AGP transfers parts of the fast but expensive graphics memory (i.e., memory containing texture data and z-buffer information) into inexpensive main memory and provides an extra access path (the AGP bus) for the graphics controller to this memory. The AGP bus is 32 bits wide, clocked with 66 MHz and data transfers are possible on the rising and the falling edge of the clock. The resulting theoretically possible transfer rate is 528 MBytes/s. The graphics controller is allowed to access the main memory part and its own private graphics memory simultaneously.

Essentially, there are three reasons, why employing AGP as physical connection between CPU and accelerator subsystem is not a feasible solution:

[12] The new specification PCI 2.1 defines a maximum clock rate of 66 MHz and a width of 64 bits effectively quadrupling the maximum theoretical transfer rate to 528 MBytes/s.

1. AGP is not expandable. The electrical specification allows only one device per AGP bus. That means, that *another* AGP bus had to be integrated into the system. Apart from the additional electrical circuitry this increases the system's arbiter complexity which would have to arbitrate main memory accesses of the CPU, PCI devices, AGP bus connected to the graphics controller and the second AGP bus connected to the accelerator subsystem.

2. The intention of AGP of moving parts of the graphics memory into the main memory is not adequately adaptable to the accelerator component. There are no parts of the coprocessor's private memory (fast, static RAM, if a DSP is used) which are worth to transfer into the main memory. Additionally, although a high transfer bandwidth is desirable, the possible transfer bandwidth of AGP exceeds the requirements of the accelerator by far.

3. Many functions defined in the AGP specification are not necessary for the connection between CPU and accelerator subsystem. For instance, transaction classes, pipelined transactions and multiple priorities for transactions are not needed.

2.2.2. Peripheral Buses

This section discusses the applicability of peripheral buses as connection mechanism between CPU, main memory and accelerator subsystem. By the term *peripheral bus* we denote a bus which does not have a direct connection to the system's main memory and CPU. Instead, it is connected to them via the local I/O bus.

2.2.2.1. Small Computer System Interface

The Small Computer System Interface (SCSI) bus is a well-established protocol for data transfer between peripheral devices and the host [Sch98] [Ame98]. The following reasons strongly suggest its usage as connection mechanism for an accelerator.

- SCSI is a well-defined and widely accepted standard. There exist controllers for virtually every computer platform.

- The (synchronous) protocol supports transfer rates up to 80 MBytes/s.

- SCSI defines 5 classes of devices, among them a so-called *multi-media device*. The latter seems suitable for an accelerator device.

The disconnect-reconnect mechanism (decoupling of command issue and respective command execution) increases bus throughput but introduces timing estimation problems.

On the contrary, several reasons contradict the usage of SCSI for our purposes: First, the computer system requires a SCSI controller to communicate with the accelerator which increases cost (whereas PCI is integrated in current PC systems). Second, SCSI is a complex protocol; every SCSI device must implement a mandatory set of commands. Again, this increases costs in comparison to simpler communication protocols. Third, up to now, no DSP hardware with SCSI has been developed so far. All DSP boards currently available rely on ISA, PCI or serial interfaces.

Concluding, it might be worth to investigate SCSI as communication mechanism but it is beyond the scope of this project.

2.2.2.2. Universal Serial Bus

At a first glance, the Universal Serial Bus (USB) seems the ideal connection mechanism between host and accelerator [USB96]. On closer examination this is not the case, because:

- The provided bandwidth is too low (maximum: 1.5 MByte/s simplex).
- The protocol is very complex, the host polls all peripheral devices in a round-robin fashion.
- USB devices have to be (very) cheap.
- The USB specification defines a maximum current consumption of 500mA per device.

2.2.2.3. IEEE 1394

The so-called "FireWire" bus was developed by Apple in 1995 and is now IEEE standard 1394 [KW95], [Sch95]. The standardization ensures a well-defined road map of further development and compatibility. The FireWire bus is intended to connect low-cost consumer audio and video devices (camcorder, set-top boxes, HDTV) as well as storage devices (hard disks, DVD player) requiring high transfer bandwidth to personal computers and to each other.

Technically, it is a 6-wire serial bus system which allows heterogeneous topologies. A maximum of 62 devices can be connected to a FireWire bus. The maximum achievable transfer rate is currently 25 MBytes/s, but the standard road map already defines 50 MBytes/s and 100 MBytes/s. The bus is auto-configuring and devices can be hot-plugged. Besides the asynchronous data transfer, isochronous transfer is possible, that is, a certain transfer bandwidth can be guaranteed for every node, as long as the sum of the bandwidths guaranteed does not exceed the overall bandwidth of the bus [För98]. As a drawback, there is no automatic error correction within the physical layer, so transfer errors have to be identified and corrected by software.

Currently, the bus is not widely accepted due to the unavailability of a broad range of suitable consumer devices and adequate interface adaptors for personal computers. It is expected that IEEE 1394 will drastically improve its market share in the near future.

2.2.3. Communication Mechanisms

Relentless of the underlying bus protocol, a DSP plug-in board may communicate with the host using one (or several) of three different communication mechanisms: mailboxes, shared memory and busmaster DMA. Their understanding is crucial for the discussion in the following chapters.

2.2.3.1. Mailboxes

Mailboxes are the simplest communication mechanism apart from serial interfaces. A mailbox consists of a single 8 to 32 bits wide word mapped into the address spaces of the host's CPU and the DSP. To ensure mutually exclusive access for both processors two hardware semaphores (one per direction) are used which must be acquired before and released after accessing the mailbox.

Data transfer using mailboxes is usually slow and CPU-intensive because for every transferred word the following mechanism must be used (transfer direction from CPU to DSP):

1. The CPU tests and acquires its respective hardware semaphore.
2. The CPU writes a word to the mailbox.
3. The CPU releases the semaphore.
4. The DSP tests and acquires the semaphore.
5. The DSP reads the word.
6. The DSP releases the semaphore.

These operations take place in strict sequential order. State changes of the mailbox and semaphores can be observed by polling or are signalled by interrupts. Because of its simplicity and the low host bus load the determination of mailbox communication times is easy. This can be an advantage under real-time constraints.

2.2.3.2. Shared Memory

Communication using shared memory is more efficient than single mailboxes. It consists of a contiguous memory region of 1...64 KBytes mapped into both processors' address spaces (so-called dual-ported memory). To synchronize accesses, hardware semaphores as described for mailboxes are used again. The size of the memory region drastically lowers the overhead necessary for accessing the semaphores.

Disadvantageously, dual-ported memory is very expensive; so board manufacturers keep the shared memory region small. Communication via shared memory can be fast and efficient but requires CPU intervention.

2.2.3.3. Busmaster DMA

With the introduction of recent buses (i.e., PCI [SA95]) a new possibility of communication between plugged-in hardware and main memory emerged, the busmaster DMA. The mechanism works as follows (transfer direction from host to DSP):

1. The host allocates a buffer suitable for DMA transfer and fills it with data to be transferred.
2. The host signals the DSP that the buffer is ready for transfer (e.g. by mailbox communication or by interrupt).
3. The DSP programs the PCI controller and defines necessary parameters (start address and length of the host buffer, transfer direction).
4. The DSP initiates the DMA by writing a certain status register of the PCI controller.
5. The PCI controller transports data from the host buffer into its own FIFO transparently to the CPU.
6. When the FIFO is filled the DSP receives a signal (e.g., an interrupt) to empty the FIFO.
7. The DSP empties the FIFO by copying the data into an appropriate receive buffer. Simultaneously, the PCI controller again fills the FIFO with new data provided there is space left in it. This proceeds until the end of the host buffer is reached.

The dedicated transfer hardware, the higher degree of parallelism as well as the invisibility to the CPU guarantee a superior performance of this mechanism in comparison to mailboxes. On the other hand, the necessary overhead per transaction is comparatively high and an initiated transfer cannot be interrupted. Therefore, for short messages mailboxes could be advantageously.

2.3. Realtime Operating Systems for DSPs

In this section we discuss some selected realtime operating system architectures which might or might not be useful for the system to be constructed.

First, a typical example of a hard realtime system (which comprises much more than a operating system) is examined briefly. We argue that we should not base our system on a hard realtime system.

The vast majority of operating systems for DSPs are very small executives with a single target application area. Adaptions and modifications to these kernels are very expensive due to their specialization. Of the remaining more universal kernels, two examples are discussed in some detail. Following, we have a short look on operating systems for DSP Clusters. The section concludes with some remarks on DSP operating system modeling.

2.3.1. MARS

A classic realtime system is MARS (MAintainable Realtime System) [KFG+92]. The authors describe a fundamental methodology for design, implementation, testing and verification of real-time systems. The project has contributed to realtime system research in numerous ways.

Basically, MARS is a distributed system of independent computing nodes which are connected via a network. It favors a consequent time-triggered approach to eliminate overload conditions under heavy system load. Advanced techniques for guaranteeing fault-tolerance to a very high degree are implemented. MARS employs a global time base and off-line scheduling. At runtime, all nodes perform fixed-interval input, computing, and output phases. A sophisticated application programming model ensures the exact determination of execution and communication times [KFG+93].

Although it might be interesting to borrow single concepts from MARS and adapt them to our needs, it is surely overkill to adapt the system as a whole. Its emphasis on fault-tolerance, distributed computing, the focus on *hard* realtime systems and its sheer extent render MARS unsuitable as base of an accelerator operating system. Generally spoken, the focus in hard realtime systems is on reliability, whereas our accelerator subsystem can be characterized as soft realtime system focusing on efficiency and minimal cost.

2.3.2. SPOX

The SPOX kernel by Texas Instruments[13], is a comparatively widely-distributed DSP operating system [Kea96]. It has been ported to different DSP processors and even to the Intel x86 platform as IA-SPOX [Lap95]. SPOX consists of a number of libraries providing (besides other) the following abstractions:

- Inter Process Communication (IPC) via mailboxes[14],
- static, priority-driven scheduling,
- synchronization with semaphores,
- buffer memory management,
- I/O using a stream concept.

Essentially, SPOX relies on the concept of coroutines for parallelism. SPOX is a static kernel; the number of DSPs tasks to execute, have to be known at compile time. The necessary functionality is statically linked to all DSP application tasks. The resulting binary image is then transferred to the target processor and started. Dynamic changes to the task set require an invocation of the compiler and linker, a DSP reset and the transfer of the newly-generated image.

Scheduling in SPOX is only possible on entry of a system call. To ensure a fair distribution of processor time, cooperation of all tasks is mandatory.

SPOX is highly adaptable to different target processor configurations using a "stacked driver" concept.

[13] The company which developed SPOX in the first place, Spectron Microsystems, has been bought by Texas Instruments in 1996.

[14] These mailboxes are not to be mistaken for the *communication mechanism* mailbox.

SPOX is distributed without source code. Therefore, exact knowledge about the timing of selected system services is hard to obtain.

[KKS94] and [Spe96] describe the SPOX kernel in more detail.

2.3.3. C80 Multitasking Executive

Designed as a high-end processor, Texas Instruments' TMS320C80[15] features a 32-bit RISC master processor with floating-point support and four independent 32-bit fixed-point DSPs integrated onto one chip [Tex97, Tec96]. Additionally, a separate video controller, a transfer controller and 50 KBytes of on-chip SRAM are provided. It was introduced in 1995.

According to [Yiu98], the C80 was designed as coprocessor for the acceleration of computing-intensive video tasks, that are

- compression and decompression of video and audio signals for video conferencing and multimedia,
- image warping, filtering, enhancement and feature recognition,
- optical character recognition and document capture,
- 2-D and 3-D graphics.

To master the vast complexity of the processor and to divide its processing power between several user tasks, a special operating system called *Multitasking Executive* has been developed by Texas Instruments [Tex95, Yiu98]. Its global structure is depicted in figure 2.3.

Client programs running at the host communicate via stubs with the MVP. They send request messages, which are received by a Message Manager running on the Master Processor (MP). The Master Processor analyzes the requests and issues commands to the four independent DSPs, called Parallel Processors (PP), which are only able to run one program at a time.

The main goal of the kernel is to provide a uniprocessor-like interface to the outside world, to give the application programmer a chance to control the complexity of the MVP. For host process acceleration the solution has the following disadvantages:

- No real-time support has been integrated.
- The current implementation does not allow the dynamic download of individual tasks at runtime.
- The kernel is implemented in C code which eases porting to different architectures but wastes valuable resources.
- The kernel offers functionality which is not necessary for host process acceleration (e.g., a complete message passing interface).

[15]The processor is often referred to as Multimedia Video Processor (MVP).

Up to now, the C80 concept of signal processing has only very limited success. The main reasons are:

1. It is believed that the programming model of the chip is simply too complex to master. The 5 independent processors form a heterogeneous multiprocessor system for which complex software is nearly impossible to design, develop and debug.

2. The price of the chip is too high[16] for a broad integration into standard PC architectures.

3. There is constant rumor in diverse forums of Texas Instruments giving up support and further development for this processor family in favor of the newly introduced C6x architecture.

2.3.4. Operating Systems for DSP Clusters

One of the main application areas for DSPs are large clusters which synchronously process streams of scanned data. Complex algorithms are partitioned into subtasks which can be executed by a single DSP. Often, several data streams have to be processed simultaneously. Therefore, twodimensional DSP arrays are a typical processor topology. The Butler kernel [WK96], developed by Perimos GmbH, shall serve as an example of a operating system tailored to such systems.

The main design criterion for Butler is the a priori knowledge of all tasks, their starting times and resource consumption. As a consequence, off-line scheduling of the DSP tasks is sufficient. The kernel provides a single time-critical *foreground task*, which is interrupted only if communication has to be performed, and an arbitrary number of *background tasks* which are scheduled only if the foreground task has been satisfied. The background tasks are assigned uncritical lesser-prioritized functions. Similar to the solution described in this work, only periodic tasks with a fixed execution time are regarded. But, contrasting, dynamic task allocation is not possible (and not necessary) in Butler, because the entry times of all tasks are known in advance. Synchronization is only necessary for foreground tasks located on different processors and is achieved by communication.

Traditional operating system functionality as semaphores, wait queues etc. are omitted in Butler. It provides some abstractions necessary for multiprocessing and communication, namely tasks, handlers and ports. The system is structured in several layers. The designer claim that it is only necessary to adapt the downmost layer to port Butler to a different DSP hardware. Up to now, Butler is available for Analog Devices' SHARC DSP.

---

[16] As for May 1998, the price for one TMS320C80 processor ranges from \$280 ...\$390 depending on packaging, clock frequency and quantity.

2.3.5. Modeling DSP Operating Systems

A group of researchers at Carnegie Mellon University concentrates on evaluating and modeling of DSP operating systems which are suitable for accelerator hardware.

In [KKS94], Mwave/OS, VCOS[17] and SPOX executed on three different DPS are briefly described and evaluated as prospect operating systems for DSP-based accelerator hardware. The authors demonstrate that the raw execution time for a given algorithm is not the main decisive factor when comparing processing with and without an accelerator. Instead, processing costs for I/O and operating system services, both of which can be kept very low for well-designed accelerators, must be considered also. Secondly, the authors propose to run any of the DSP kernels at an accelerator board without publishing performances numbers.

In [SK94] the authors describe their modeling methodology on a high-level basis. First a classification of real-time systems is developed. Following, IBM's Mwave/OS DSP operating system[Lap94] is analyzed and the scheduling timing is modeled. The authors claim deviations between modeled and measured times smaller than 1%. Using the classification in [SK94], the system described in this work is a timer-driven, preemptable realtime operating system.

The modeling methodology is further detailed in [KKS95]. In the first part, the authors present analytic equations to derive execution timing for different classes of scheduling algorithms. The basic idea is to decompose the scheduling mechanism of a given DSP operating system into a set elementary components whose execution times can be measured or obtained by the manufacturer (i.e., time to preempt a running process, time to identify the next process to run etc.). By using the modeling equations, it is then possible to model the overall scheduling overhead in a very accurate way.

The second part of the paper briefly describes the three analyzed DSP operating systems (Mwave/OS, VCOS and SPOX) as in [KKS94] and explains the methodology to obtain the model equations. To validate the models, synthetic task sets were used to estimate the *breakpoint untilization*[18] The authors claim a worst case difference of 5% between analytically obtained and measured times.

The final part evaluates the analyzed operating systems and further details some design aspects of Mwave/OS. It is interesting to note that for the given task sets, SPOX was completely (100%) consumed by its overhead whereas VCOS and Mwave/OS revealed a system overhead of 8% and 21% respectively.

2.4. L4

To understand the derivation of system structure in chapter 3, some knowledge of the microkernel L4's key concepts is a prescription. [Lie95] briefly discusses the main features of L4, a detailed description of L4 is contained in [AH98] and a more formal description

[17] VCOS is a DSP operating system developed by AT&T for their DSP3210 in 1993. When Lucent Technologies overtook AT&T Microelectronics, support for VCOS was discontinued.

[18] The point at which further increase in processing requirements causes a process to miss its deadline.

of processor-dependent and hardware-independent aspects of L4 can be found in [Lie96].

L4 has been developed consequently adopting the minimality principle: "More precisely, a concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality." ([Lie96], pg. 7). It provides the following abstractions:

- address spaces,
- threads,
- inter-process communication (IPC),
- Clans & Chiefs.

Threads are activities in an L4 system. A thread is associated with a memory space and a set of registers. To identify threads, unique identifiers are used. Threads communicate with each other using the IPC mechanisms described below.

Address Spaces

An address space is a set of memory pages a thread is allowed to access. L4 provides three operations, *grant*, *map* and *flush* to construct address spaces on top of the initial address space σ_0 (the physical memory). A *grant* operation removes a pages from the address space of the granter and includes it into the address space of the grantee. A *map* includes a page into the mappee's address space without removing it from the mapper's address space (mapper and mappee are now able to access the page). A *flush* operation removes a previously mapped page from the mappee's address space.

These operations enable the concept of so-called *external pagers*, that are user-level processes which manage address spaces of other processes. The *root pager* manages the physical address space σ_0.

External pagers are used as follows: A process accesses a non-mapped memory page. This access triggers a page fault exception which is translated by the microkernel into an IPC message directed to a user-level pager. The pager finally resolves the page fault by mapping the appropriate memory page. This concept provides unsurpassed memory management flexibility. As an example, it allows to construct hierarchies of pagers.

Inter Process Communication (IPC)

Inter process communication is the only way of data transfer across different address spaces. Communication in L4 is synchronous and unbuffered, requriring an agreement between sender and receiver. Data is exchanged via:

- registers,
- direct or indirect strings,
- *flexpages*, that are contiguous regions of a virtual address space.

Direct strings are contained within a message, whereas indirect strings are described within the message using a so-called message dope. Flexpages are exchanged by reference using flexpage descriptors.

Communicating threads can define a maximum time to wait for the successful completion of a communication operation, the *timeout*. Hardware interrupts are trapped by the kernel and converted into empty IPC messages. Due to the highly-optimized IPC mechanisms, L4 is considered the fastest microkernel available [LES+97].

The concept of Clans & Chiefs is irrelevant for this work, and therefore not described further.

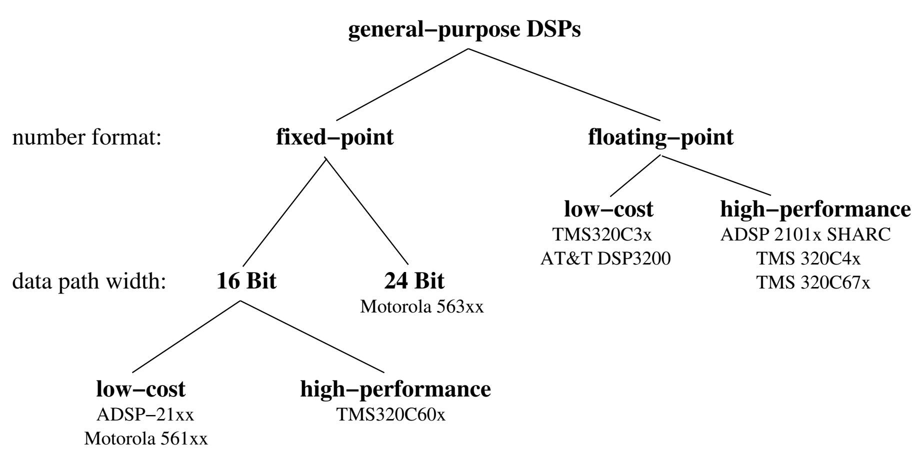


Figure 2.1.: Systematization of DSP architectures

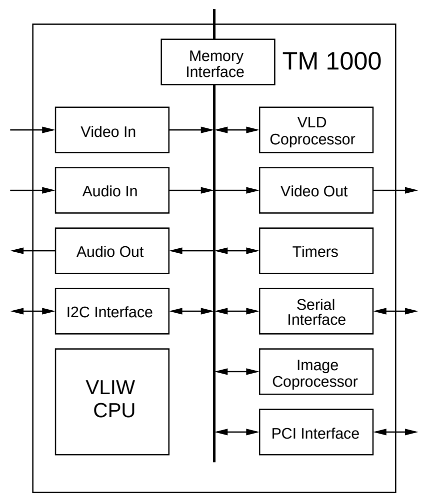


Figure 2.2.: Philips TriMedia TM-1000 block diagram

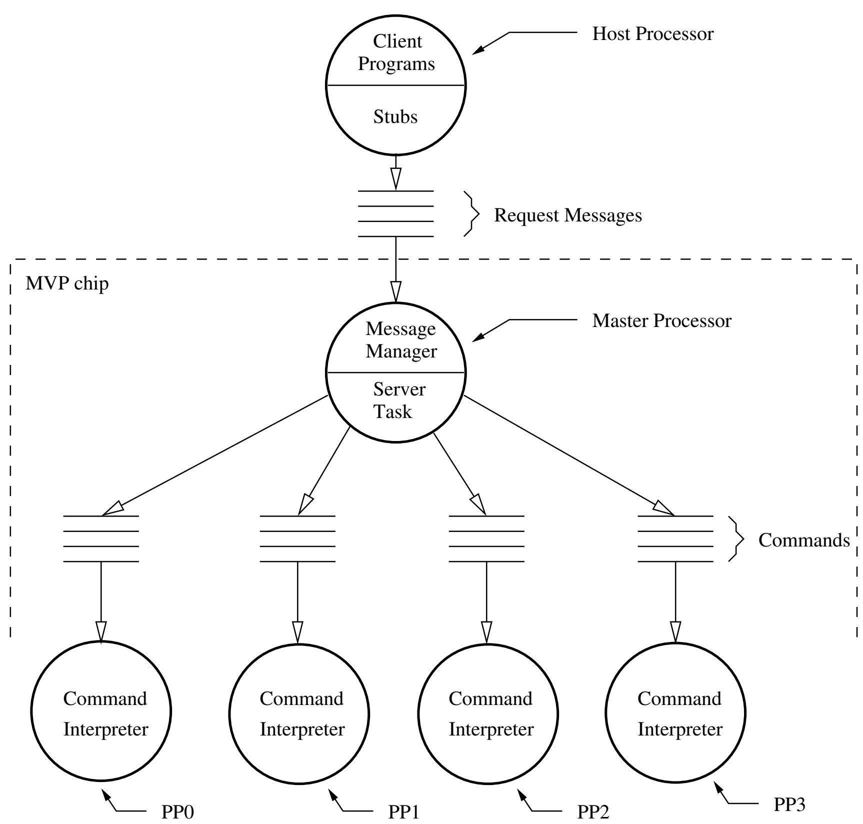


Figure 2.3.: Data Flow in C80 Multitasking Executive [Tex95]

3. Concept

3.1. Introduction

To summarize the discussion of chapter 2 the current state of the art of multimedia accelerators for personal computer systems can be described as follows: There exists a selection of suitable processors and hardware interfaces, one of the most promising are DSPs connected to the host via PCI. A suitable base for the construction of computer systems guaranteeing QoS are microkernel-based operating systems. Existing operating system software for DSPs is specialized for embedded systems and shows one or several of the following disadvantages:

- The specialization causes lack of flexibility.
- It introduces significant overhead and provides unnecessary functionality.
- None of it takes advantage of the microkernel structuring paradigma for host operating systems.
- None of it is tailored to support DSPs as accelerators.

We conclude that a fundamentally new approach has the potential of overcoming these disadvantages and allows to exploit the processing power of an accelerator subsystem to a maximum. This chapter describes the proposed system structure. First, the requirements for the system to be constructed are formulated. Afterwards, the system's architecture is derived. The following sections discuss the components comprising the host part and the accelerator part in detail. Finally, the main features of the constructed system are summarized.

Throughout the work we use the term *accelerator subsystem* to denote the built-in board with the DSP, its private memory and assorted periphery. The terms *accelerator* and *DSP* are often used synonymously and refer to the processor of the accelerator subsystem. The *accelerator server* is a user-level process on top of the microkernel executed at the host. By *guaranteed acceleration* we emphasize that an acceleration service granted by the DSP must not degrade its service quality until the end of this service, which is determined by the client. The term *accelerator process* identifies a DSP process which provides a particular acceleration service to an application process running at the host. Section B of the appendix contains a concise glossary.

3.2. Discussion of System Structure

The basic requirements of the system to be constructed can be briefly formulated as follows:

1. The system integrates an accelerator subsystem into a microkernel system running at the host processor to provide acceleration of certain services to applications running at the host. Two microprocessors, the host's CPU and the accelerator's main processor, which interact closely, are present in the system.

2. The solution provides *guaranteed* acceleration, i.e., if the accelerator is able to perform a requested service, the negotiated quality of service is sustained until the end of this service.

3. The accelerator subsystem is able to serve more than one application client process simultaneously (the number of clients is limited only by the underlying hardware, for instance, communication bandwidth between CPU and accelerator's processor or the processing power of the accelerator's processor).

To fulfill these requirements several different system architectures are conceivable. This section discusses the alternatives and argues that the most suitable is a microkernel server executed at the host in combination with a small kernel running on the accelerator's processor. As real-world example, we rely on L4 by Jochen Liedtke [AH98, Lie96]. Knowledge of basic L4 services and concepts is presumed.

3.2.1. Transparent Threads

The first and obvious approach of a system meeting the requirements stated above uses *transparent threads*[1]. A certain set of threads is executed by the host CPU and another set of threads is executed by the accelerator. From a user application process' point-of-view, threads executed by the CPU need not be distinguished from threads executed by the accelerator, only the microkernel is able to distinguish the location of a particular thread. It is evident that the microkernel must be extended to manage the accelerator subsystem accordingly.

The obvious major advantage of this approach is that application processes which use acceleration only require minor modification[2]. Of course, such a solution is preferable to all which require complex application modification. The resulting system structure is depicted in figure 3.1. The microkernel directly controls the accelerator subsystem.

The main obstacle to providing this abstraction is the absence of a virtual memory concept of DSPs: they do not incorporate a memory management unit (MMU). Hence,

[1] The term *transparent* refers to communication. The binary images of transparent threads differ due to different instruction sets of both processors.

[2] A certain amount of modification is inevitable, due to specific differences in processor architecture (e.g., certain DSP addresses are mapped to peripheral devices, whereas the host CPU utilizes a different address space for I/O).

the microkernel's physical address space consists of *two* distinct heterogeneous memory spaces located on different processors. Only one of this spaces is managed by a MMU.

A first approach to overcome this flaw is to design accelerator threads compatible to microkernel threads. As a consequence, all system calls may be executed by accelerator threads. This introduces serious problems as discussed in the following:

The address space operations of L4, *grant*, *map* and *flush* (cf. section 2.4), must be adapted. The *map* operation between threads located on different processors could only be implemented as follows: The kernel copies the page physically to the mappee[3]. Then, all subsequent write accesses to this area must be tracked by both the mapper and the mappee and propagated on the respective other processor to keep consistency of both copies. This has to be sustained until the mapping is revoked. Although this mechanism seems feasible for a few mappings, it is clear that it introduces serious overhead[4] if the number of established mappings exceeds a certain limit. Similar problems arise for *grant* and *flush*.

Furthermore, the concept of external pagers appears hard to adapt to transparent threads executed on the accelerator. Note for instance, that the root pager has control over two completely independent address spaces (one for each processor), which are in addition heterogeneous (depending on the concrete DSP type).

Another approach is to run a simplified thread package on the accelerator and limit compatibility of microkernel and accelerator threads (e.g., not allow a *map* between threads located on different processors). The following two sections discuss this approach based on thread scheduling and inter-process communication (IPC).

Thread Scheduling

In principle, two variants to schedule transparent threads are imaginable:

One Scheduler. The microkernel scheduler schedules *all* threads existing in the system, in other words it has control over both processors. Of course, a different scheduling mechanism has to be implemented for the accelerator threads, otherwise service guarantee is hard to achieve. Clearly, it can only signal to switch execution to a different thread. The context switch has to be performed by the accelerator itself. This variant seems practicable but increases the scheduler's complexity.

Two Schedulers. Threads executed at the accelerator are scheduled independently from threads executed at the CPU. This requires scheduling functionality on the DSP. The resulting system architecture seems to be very complex (e.g., the adaptation of thread-related system calls) and a certain amount of the DSP's processing resources must be dedicated for thread management.

[3] To clarify the relationship between the two involved threads we use the same terminology as Liedtke in [Lie95], e.g. mapper – mappee.

[4] The overhead is introduced by a) the copy operation, b) the continuous monitoring of write accesses and c) the enforcement of the changes on the respective other side.

Again, the main problem results from the two separate address spaces and the non-existing notion of virtual memory in DSPs. A thread creating a task to be executed on the respective other processor must provide among others a pager for the thread to be created.

IPC

Regarding inter-process communication, the interesting case is IPC between threads executed on different processors. Passing data between threads by reference using *map*, *grant* and *flush* has been already discussed in the first paragraphs of this section, so we can concentrate on passing data by value (i.e., as string or as in-line data).

As an example, a thread executed on the CPU issues a system call to send a string message to a thread executed on the accelerator. After receiving the system call, the microkernel must recognize that this is not an ordinary IPC between CPU threads. Instead of performing the necessary copying operation it must transmit the contents of the message to the accelerator using the appropriate hardware mechanism. This necessitates the introduction of a special hardware communication mechanism into the microkernel, which strongly contradicts the microkernel paradigm.

Another problematic aspect is service guarantee: How is the actual workload of the accelerator determined and who decides whether to grant or reject a service request of an application? Using *transparent threads*, a special server is necessary for admission control, which is consulted by the microkernel when a new thread is started on the DSP. But if a server is needed anyway, why not locate *all* DSP-related functionality within this server instead of splitting it between admission control server and microkernel? We will follow this idea in section 3.2.3.

Third, IPC between two threads executed by the accelerator entails data transfer between DSP and CPU, because the system call is executed by the microkernel, although the DSP is in principle able to perform the data transfer autonomously. Overhead is introduced.

In summary, we can conclude that transparent threads lead to a high implementation complexity and most likely to a slow-down of essential microkernel services, and therefore, are an unsuitable abstraction for integrating an accelerator into a microkernel-based system.

Besides, we can conclude that microkernel applications have to be modified to exploit acceleration functionality in a microkernel environment, because transparent threads are the only possible system structure to avoid application modification.

3.2.2. A Microkernel Server

Obviously, the idea of transparent threads leads to a much too complex system structure. The easiest way to drastically reduce this complexity is to treat the accelerator subsystem as a conventional peripheral device. The accelerator subsystem is controlled by a microkernel server, which acts as a conventional device driver (cf. figure 3.2). The global interaction between an application client process, the server controlling the accelerator subsystem and the accelerator is realized as follows:

If an application client process requests acceleration, it must first find out which services are offered using a name service provided by the server. It then requests a particular service. Presumed, the DSP is not used by another client, the server downloads the adequate DSP program to the DSP and starts it. Afterwards, the server is responsible for data transfer between the client and the process executed on the DSP.

The advantages of this approach are substantiated by its simplicity: No control overhead is paid on the accelerator's side, all its processing power can be dedicated to client acceleration. The host server structure is simple and its processing requirements are expected to be negligible. The communication between application client process and the accelerator is managed easily.

The disadvantages mainly result from the reduction of the acceleration subsystem to a dumb peripheral device, its flexibility is not exploited: Only one function at a time can be accelerated. A possibly simple function excludes all other clients from the acceleration service. Depending on the workload at a given time and the processing capabilities of the concrete accelerator's processor most of its resources are wasted. This is a contradiction to the requirements stated at the beginning of this section and therefore unacceptable.

Another disadvantage is that applications have to be modified to exploit acceleration[5].

Finally, approaches with this architecture have been published before. The work described in [Bau95] and [BH96] integrates a DSP subsystem into the VSTa microkernel. Köhler describes in [Köh98] two integration variants of a DSP in a monolithically structured operating system. Furthermore, several integrations of DSPs in conventional PC architectures were developed by the industry at the beginning of the 90s (for a detailed discussion on related work, we refer to chapter 2).

3.2.3. Microkernel Server and Accelerator Kernel

In section 3.2.2 we argued that it is necessary to have some kind of controlling instance running on the accelerator's processor, otherwise simultaneity of several acceleration services is not possible. In section 3.2.1 we showed that it is not possible to provide acceleration to applications without modifying them by the usage of transparent threads.

Hence, the model of a dedicated thread (the *Accelerator Server*) executed on the CPU in combination with a separate kernel on the accelerator has been chosen (cf. figure 3.3). At a first glance, its structure seems more complex than the approaches described in the preceding sections. The host microkernel runs a set of servers (X Server, Y Server, etc.) which provide traditional operating system functionality. One of this servers, the accelerator server, is responsible for the management of the accelerator subsystem. Application client processes, which request the functionality provided by the servers, reside on top of the system. The software configuration on the accelerator side consists of a kernel and a set of accelerator processes, which are responsible for the provision of

[5] However, this cannot be avoided anyway, as we have already discussed in section 3.2.1.

acceleration services.

An important issue is the division of program complexity: Is it favorable to construct a complex server and a simple accelerator kernel or a simple server and a complex accelerator kernel? Clearly, the CPU microprocessor has a general-purpose instruction set architecture (ISA), whereas the accelerator's processor (the DSP) has a specialized chip architecture and instruction set. Accordingly, general-purpose processors have a significant performance advantage for executing control-oriented code over Digital Signal Processors.

Therefore, for the whole design we maintained the following principle: **As much as possible control-oriented functionality remains on the host side. Management overhead on the accelerator's processor must be avoided at all cost.**

This principle has another inherent advantage: code development, debugging and testing is solely done on the host processor. From a developer's point of view the accelerator's processor is a remote processor which is much more difficult to debug than the CPU. The simpler the software structure executed on the accelerator subsystem, the easier the implementation, testing and debugging process for it can be kept. The resulting system is more developer-friendly in terms of maintainance in comparison with a simple host server and a very complex kernel on the accelerator's processor.

3.3. System Components

After reasoning about the global system structure this section discusses in detail both parts of the acceleration system, the host server and the necessary software on the accelerator. The first subsection is dedicated to the microkernel server executed on the host which is responsible for managing the accelerator subsystem. The second subsection discusses all aspects of the software executed on the accelerator including the kernel and the accelerator processes. The final subsection describes the interaction of both parts with each other and with clients. Beforehand, a short example of the interaction of the components is given.

Reconsider figure 3.3. An application client process running at the host may require decoding and subsequent playback of audio data. It will issue a corresponding service request to the accelerator server. The server checks, whether sufficient resources are available on the accelerator subsystem and grants or rejects the service request. Provided the request has been granted, the application client process will now establish a transmission of an audio stream between itself and an accelerator process capable of decoding the audio data format via the accelerator server (cf. figure 3.4). This accelerator process decodes and subsequently transmits the decoded audio data to a digital-to-analog converter. The service is ended at an arbitrary time by the application client process.

3.3.1. Host Part: The Server

Basically, the host is responsible for the accelerator management which consists of four different parts: name service, workload estimation, admission control and scheduling of accelerator processes, data transfer.

3.3.1.1. Name Service

An application requiring accelerator functionality must first ascertain, what acceleration services are provided by the accelerator at all. The respective functionality is provided by the *name service* component of the accelerator server. This includes means to integrate new acceleration services into the system.

More precisely, the name service provides the following functionality:

- a service for application processes to query the existence of a certain accelerator functionality,
- a function for a privileged process to register a new service (i.e., a new accelerator function to be potentially used by application client processes is integrated into the system),
- a function to de-register an obsolete service from the system.

Usually, services are identified by strings representing services, hence the server must maintain a data structure storing all provided services and indexed by name. Additional necessary information contained in this structure are

- the location of the according binary program data implementing the requested service,
- pointers to resource requirements (accelerator's processor utilization, necessary memory, seizure of peripheral devices).

3.3.1.2. Workload Estimation

Given that the requested functionality can be provided by the accelerator, the next task is to estimate the workload of the accelerator process to be installed. One basic requirement of the system to be constructed is the guarantee of the quality of service to application client process. Therefore, a new accelerator process must

1. not exceed the (remaining) processing capabilites of the DSP,
2. fit into the schedule of already established accelerator processes.

Hence, it is essential to exactly determine the workload of the new accelerator process. To simplify scheduling we limit the allowed accelerator processes to processes with constant execution time and constant activation frequency. The workload of a particular accelerator process π can then be described by two *scheduling parameters*:

$t_{e,\pi}$ the worst-case execution time of the process π per activation, and

$t_{p,\pi}$ the period of process π, the time between two consecutive activations of that process π, or (alternatively)

$f_{a,\pi}$ the activation frequency of π, the number of activations of the process π in a fixed time interval (it holds $f_{a,\pi} = \frac{1}{t_{p,\pi}}$).

The scheduling parameters of a particular accelerator process depend on

- the implementation of the algorithm on the accelerator's processor,
- the speed of the accelerator's processor,
- the data to be processed.

The parameters have to be derived by the workload estimation component of the micro-kernel server everytime an application client process requires functionality (and, consequently, an accelerator process has to be installed). For simple algorithms, for instance play-back of audio data, it suffices to generate an execution time table by measuring the execution time of the accelerator process for every possible combination of stream parameters (e.g. , word length: 8, 12, 16 bits; sample rate: 8, 12, 32, 44.1, 48 kHz). Whenever an application client process requires this functionality for a certain stream, the server extracts the processing parameters directly from the stream (e.g., by analyzing its header), uses them to index the execution time table for this algorithm and obtains the correct scheduling parameters.

For algorithms whose execution time depends on the data to be processed, a previously generated stream description must be used.

3.3.1.3. Admission Control and Scheduling

In principle, two different approaches to scheduling are possible: dynamic or static scheduling. *Dynamic* scheduling means that every time the currently executed process is deactivated (e.g., due to an unavailable resource or an elapsed processor time slice), the scheduler selects the next process to run according to its scheduling strategy. The scheduling function is invoked during every context switch. The exact succession of processes cannot be determined in advance, because the set of processes ready to be executed and their scheduling parameters constantly change. In contrast, a *static* schedule is generated in advance and the processor follows this precomputed schedule for long periods of time. This is only possible, when the scheduling parameters of processes are known in advance[6]. If a change to the process set is necessary, a new schedule has to be generated. The scheduling function is invoked in large intervals.

Several reasons strongly suggest a *static* scheduler located at the host:

- The set of processes running on the DSP is expected to be constant for very large periods of time.

[6]Static scheduling is simplified drastically, if the scheduling parameters are constants.

- The scheduling parameters of all processes are constant and known in advance.
- All processes are cyclically scheduled. The order of activation is constant.
- Dynamic scheduling at the host incurs a significant communication overhead, because for every context switch a communication to the server must be performed (the scheduler has to select the next process to run).
- The implementation complexity is much higher for a dynamic scheduler.
- A schedule recalculation is necessary only when a new process has to be installed or a process has finished execution.

For the same reasons it does not make sense to port the microkernel scheduling algorithm to the accelerator server.

A desirable side effect of a static scheduler is that the required functionality on the accelerator side is reduced to the context switch. This minimizes the negative influence of control-oriented code onto the accelerator performance.

The scheduler must be invoked every time a new accelerator process shall be installed at the accelerator subsystem (that is, an application client process requires an acceleration service). Prior to schedule generation, admission control is necessary. The host must decide, whether or not the additional accelerator process is feasible under the current accelerator workload. If this is the case, a new schedule can be generated, otherwise the service request must be rejected.

Jones et al. [JRR97] describe an algorithm for admission control and scheduling, which can be adapted to our purposes. It uses a heuristic search over a quantized representation of the DSP execution time resource. The mechanism works as follows:

First, a binary tree of arbitrary depth is constructed, the *availability tree*. The nodes of this tree represent quantities of DSP execution time, their depth within the tree corresponds to a 2^n-multiple of the base period. Initially, all nodes are free, that means the associated processing time quantum is available to processes. If a new DSP process is to be installed, its period is first quantized to the next 2^n-multiple of the base period. Then, the availability tree is traversed in search of a node with sufficient free processing time. If a node with sufficient processing time is found, its available processing time is reduced by the process' execution time and the process is associated with the node, the process is schedulable. The resource consumption has to be spread to the ancestor and child nodes.

If no node can be found, the period (and the execution time) of the process are divided by two and the traversal is reiterated. This can be repeated until the period of the process equals the base period. If still no node with sufficient processing time is available, the process is not schedulable under the current workload, its installation is rejected.

If the process was schedulable, a new schedule has to be generated. To this aim, the availability tree is traversed several times from the root to its leaf nodes such that if a branching point is reached which has been visited before, the other alternative is chosen.

The succession of nodes-associated processes constitutes the new schedule. More details of this algorithm can be found in [JRR97].

Figure 3.5 shows a very simple example with three processes A, B, and C and the kernel itself (OS). The depicted schedule executes the accelerator's processes A, B, C, B, successively, which all occupy one time slot. The last slot (OS) is reserved for the kernel. After that, the schedule repeats. As can be seen, it is possible to reserve more than one slot for a particular process within one period.

The schedule has to be transmitted to the accelerator after its generation. It simply consists of one period of the sequence of processes to be activated.

A single non-realtime process may be installed additionally. This process is activated, whenever a free time slot is encountered in the schedule. If all slots are allocated, this process is not executed, therefore no realtime guarantee is given (and no scheduling parameters t_e and t_p are associated with it). Even a processing progress cannot be guaranteed. This (aperiodic) process is not regarded by the scheduling algorithm.

Processes with variable execution time are not guaranteed to finish execution before their respective deadlines. Instead, a deadline has to be chosen, which ensures that their *average* processing requirements are fulfilled. Or, if their functionality permits it, these processes could be installed as the non-realtime process consuming all remaining processing time.

Processes with variable periods cannot be executed by the accelerator.

3.3.1.4. Data Transfer

This component is responsible for the physical communication between the microkernel and the accelerator subsystem. In principle, communication occurs in the following three situations:

1. **DSP Code has to be downloaded.** Code download occurs during startup of the system and whenever an accelerator process has to be installed. The amount of data to be transferred is small and the timing is not critical. This communication is unidirectional, directing from the CPU to the accelerator. The scheduling parameters t_e and t_p of a new process have to be transmitted, too.

2. **A new schedule has to be downloaded.** Whenever a new accelerator process has to be installed, a new schedule must be transmitted. To guarantee that the established processes are not affected, this communication has to be performed atomically within one (free) time slot of the old schedule. Its timing is critical. The amount of data to be transferred is small and constant. This communication is unidirectional.

3. **Application data is transferred.** Established accelerator processes have to transmit data to and/or from the associated application client process, communication is uni- or bidirectional. The amount of data to be transferred varies depending on the algorithm. The timing can be very critical, depending on whether or not the service needs to be performed in realtime. This kind of communication is by far the most complex and sensitive.

Current DSP plug-in boards provide at least three different hardware communication mechanisms which must be mapped to the described communication situations. Furthermore, there is a selection of recently-introduced hardware communication mechanisms which should be taken into consideration for next-generation accelerator hardware. These are discussed concisely in chapter 4.

Finally, the microkernel server needs a controlling instance, which observes and guarantees the correct sequence and interaction of the individual server components.

3.3.2. Accelerator Part: Accelerator Kernel and Application Processes

Some of the system details concerning the accelerator's side have already been discussed in the previous section. This section explains the details of the accelerator's software configuration.

3.3.2.1. Accelerator Kernel Design

As pointed out in the previous section the accelerator's schedule consists of a cyclically repeated set of time slots. One time slot is always reserved for the kernel. The size of the always reserved for the kernel. The size of the kernel time slot may be adapted by the server. On system startup, the kernel is the only process running, it can consume all processing time of the accelerator. As soon as a process is to be installed, the kernel's slot size is reduced according to the requirements of the process to be installed. The server ensures that the kernel's slot size does not fall below a predetermined limit.

Basically, the accelerator kernel is responsible for:

- context switch,
- communication with host server,
- process management,
- data transfer to and from DSP peripheral devices.

Context Switch

We rely on *preemptive* scheduling instead of *cooperative* scheduling for the following two reasons: First, with preemptive scheduling, an application exceeding its statically calculated execution time cannot affect the operation of other accelerator processes. This is necessary because for non-trivial accelerator processes it cannot be proved that the obtained execution time is not exceeded. Therefore, the kernel has to ensure that a process excessing its deadline is de-scheduled to protect other processes. Second, the necessary modifications to existing accelerator applications are less complex than for cooperative scheduling. The application does not need to execute calls to relieve the processor, only the mechanisms for data transfer have to be adapted. The disadvantage of a preemptive context switch is that always all registers have to be saved by the kernel whereas for cooperative scheduling it is possible to assign the responsibility of context saving to the application and thus save only the registers which are currently in use.

Due to the constant execution times the context switch can be initiated by a timer. Provided a timer interrupt has been issued, the associated interrupt service routine (ISR) has to do the following:

1. save the context of the process executed up to now,
2. read the next process to be activated from the precalculated schedule,
3. obtain the according deadline of this process and setup the timer accordingly,
4. restore the context of the process to be activated,
5. return from interrupt.

Because the context switch is by far the most frequently used kernel operation, an efficient implementation is crucial.

Communication with host server

As described in section 3.3.1.4, communication is necessary in three different situations:

1. DSP code has to be downloaded,
2. a new schedule has to be downloaded,
3. application data is transferred.

As we have argued in section 3.3.1.4, it is mandatory to perform the transfer of a schedule within a single kernel time slot. This is reasonable, because the amount of data to be transferred is small[7] and constant, hence the necessary transfer time is also small and constant. The following mechanism is proposed: Whenever the kernel is activated it first checks, whether the host wants to transfer a new schedule. If this is the case, the transfer is initiated immediately. The received schedule overwrites the obsolete schedule to avoid copying overhead. Then, the kernel executes other services (e.g., application and code data transfer). When the timer interrupt finishes kernel execution, the interrupt service routine uses the newly-installed schedule to identify the next process to be activated.

In principle, an accelerator process is allowed to transfer data for every activation. Hence, data transfer for application processes also has to be finished during one kernel slot. The transfer rate of the underlying hardware data transfer mechanism directly determines the size of a buffer to be transmitted (i.e., the faster the mechanism works, the bigger the communication buffers to be transferred can be chosen). As pointed out earlier, a very efficient hardware transfer mechanism is a must. Furthermore, the size of the communication buffers must be constant, to guarantee a constant application data transmission time.

[7] The schedule simply consists of a sequence of process numbers of one word each.

Due to different program sizes, the time needed for application code download varies considerably. Therefore, code download cannot be limited to finish within one kernel time slot. It follows that the used hardware communication mechanism has to be interruptible[8]. Code download is performed simply within the remaining time of the kernel slot and may span several kernel activations. This introduces no problems, because the guarantee of a certain quality of service starts with the *activation* of the application process (i.e., the download of the new schedule) and not with its *installation*.

In section 3.3.2.1 we pointed out that the size of the kernel slot can be adapted by the scheduler. The *minimum* kernel slot size is determined by the following time constraints:

$$t_{k,min} = t_{dl,sched} + n(t_{dt,to} + t_{dt,from}) + t_{rest}$$

with

$t_{k,min}$	minimum size for the kernel time slot
$t_{dl,sched}$	time for scheduler download
n	maximum number of different installable processes
$t_{dt,to}$	time for transfer of one data buffer to the process
$t_{dt,from}$	time for transfer of one data buffer from the process
t_{rest}	time for miscellaneous control.

Process Management

Process Management is straightforward because almost all management is done by the host server. The accelerator simply has to maintain a data structure which maps the process numbers in the schedule to the necessary process information. It comprises:

- the size of its time slot (i.e., the value to be written to the timer),
- the contents of the instruction pointer,
- the contents of the stack pointer,
- the start addresses of the associated data buffers (input and output).

Data Transfer to and from DSP Peripheral Devices

By the term DSP peripheral devices we denote interfaces to the outside which are connected to the accelerator's subsystem (instead of being a part of the embedding computer system, e.g., a sound card). Within the context of this work, we focus on analog-to-digital converters (ADC) and digital-to-analog converters (DAC).

Common to DSP peripheral devices is their high access frequency: a DAC playing back an audio data stream of CD quality (sampling frequency of 44.1 kHz) must receive a new data value approximately every 22 milliseconds. An accelerator process controlling this device has to be activated every 22 milliseconds and the granularity of the scheduling (i.e., the size of the base scheduling time slot, cf. section 3.3.1.3) has to be scaled down

[8] Hence, PCI Busmaster DMA is *not* a suitable mechanism for code download.

to this value. This leads to a long schedule and very frequent context switches as soon as such a process is installed. The accelerator's efficiency is decreased.

Therefore another approach is proposed: Two small routines are provided by the kernel. One routine reads a value from the analog-to-digital converter and stores it into a predefined buffer if activated. The other routine reads a value from another predefined buffer and writes it to the digital-to-analog converter on activation. Both routines are installed on demand as interrupt service routines by the accelerator server. They are sufficiently small (approximately ten instructions) so that their invocation does not influence the timing of the concerned accelerator process. As a result, they can be called any time, provided they do not modify the contents of any register. The accessing process configures the timer[9] with the according activation frequency and informs the ISR about the location of the data buffers. The accelerator server ensures that only one accelerator process has access to the converters.

A better solution does not need the described mechanism but requires hardware support: converters with private buffers and separate timers. Then the (periodic) activation of the converter is configured during the initialization phase of an accelerator process. Later the accessing process must only guarantee that the buffers are always filled. Such a hardware solution is highly desirable but beyond the scope of this work.

3.3.2.2. Accelerator Applications

The system to be constructed focuses on the acceleration of selected algorithms. Common to them are the following characteristics:

First, we concentrate on *stream-based* operations, because

- they are typical for multimedia algorithms,
- they are a cause for performance degradation in conventional personal computer and operating system environments due to their considerable resource consumption,
- the architecture of DSPs is tailored to stream-based algorithms.

Second, the applications to be run on the accelerator all have *constant periodic activation* and the *execution time per activation is constant* or varies only to a very low degree. This allows static scheduling and, especially, the utilization of the low-overhead scheduling algorithm described in section 3.3.1.3.

Third, the implementation of the algorithms executed by the accelerator must be suited to its instruction set architecture (ISA), otherwise acceleration is unlikely. This means the accelerator's implementations have to be executed significantly faster than an implementation for the CPU. Design guidelines for the implementation of efficient DSP code are given in section 5.2.3.

The following list offers some application examples which fulfill the formulated characteristics and, hence, are candidates to be implemented for the prototype.

[9] As a prescription the used DSP must have another timer available.

- MPEG Audio coding and decoding,
- JPEG still picture coding and decoding, especially when used in Motion JPEG (MJPEG) streams,
- Music synthesis,
- cryptographic algorithms.

The methodology for the development of an accelerator's application can briefly be described as follows:

First, an application is chosen based on the characteristics described above (stream-based operation, constant periodic activation, constant execution time). Then, the algorithm is implemented (the provided data transfer mechanisms of the accelerator kernel have to be exploited) and tested. Following, the application is benchmarked via a simulator tool and its execution parameters are obtained. Finally, a ready-to-download binary file is generated and the obtained resource requirements and execution parameters are integrated into the database located within the accelerator server.

Existing applications have to be modified such that they utilize the data transfer mechanisms provided by the kernel instead of directly transmitting data to the host or accessing an analog-digital converter.

During the system's runtime, it is necessary to download to and start accelerator code from different addresses, depending on whichever accelerator memory region is unused at the moment of application download. A simple solution would be to generate relocatable code and determine the starting address immediately before code download. Unfortunately, common DSP development tools do not support relocatable code generation. A feasible solution would be to invoke the code generation tools on demand by the accelerator server with a dynamically-determined starting address as parameter. The third possibility is to pre-generate a number of binaries per application which are linked to different memory regions and trust that at least one of these memory regions is available when the application process is to be installed (of course, no guarantee can be given). Eventually, both methods can coexist.

3.4. Summary of Architecture

This concluding section summarizes the main architectural features of the designed system. In comparison to existing solutions, the system has the following attributes:

- It is possible to use a DSP as accelerator for selected algorithms within a standard PC architecture running a microkernel environment.
- It is possible to execute several processes on the DSP simultaneously (provided that they do not simultaneously need identical DSP system resources, like converters).

- Accelerator functionality is dynamically adaptable (i.e., can be modified during system's runtime). Installation of a new DSP process (code download, schedule modification, etc.) does not affect already executing DSP processes.
- The number of installable processes is variable and only depends on the processing capabilities of the DSP.
- The DSP processes must have a periodic activation and a constant execution time.
- Additionally, one DSP process which is aperiodically scheduled may be installed.
- The execution time of all periodically scheduled processes is guaranteed.
- The processing capabilities of the DSP are exploited to the maximum, because its management overhead has been consequently minimized.

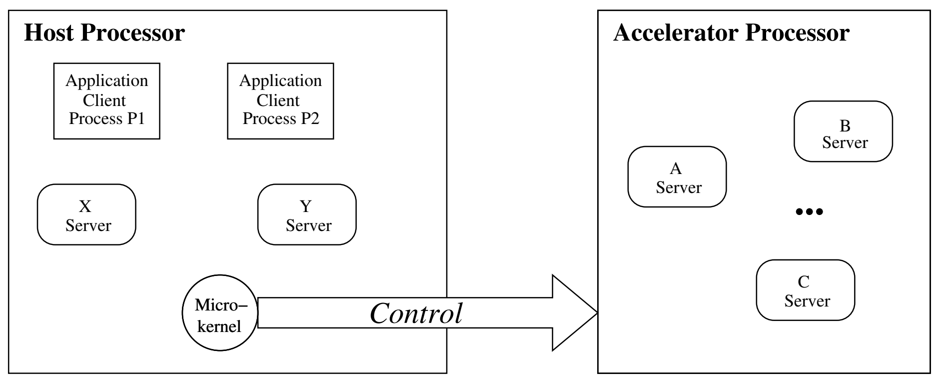


Figure 3.1.: System Structure with Transparent Threads

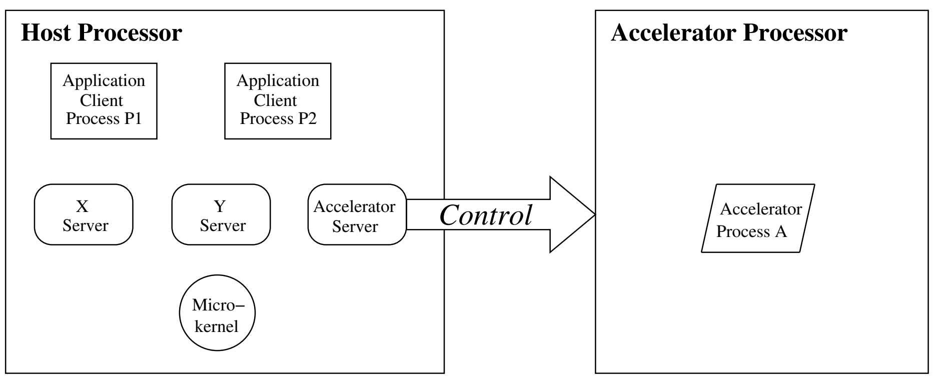


Figure 3.2.: System Structure with a Microkernel Server only

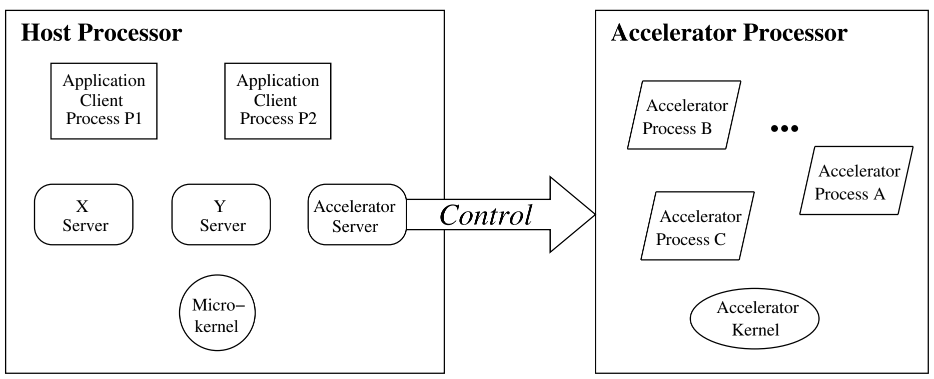


Figure 3.3.: System Structure with Server and Accelerator Kernel

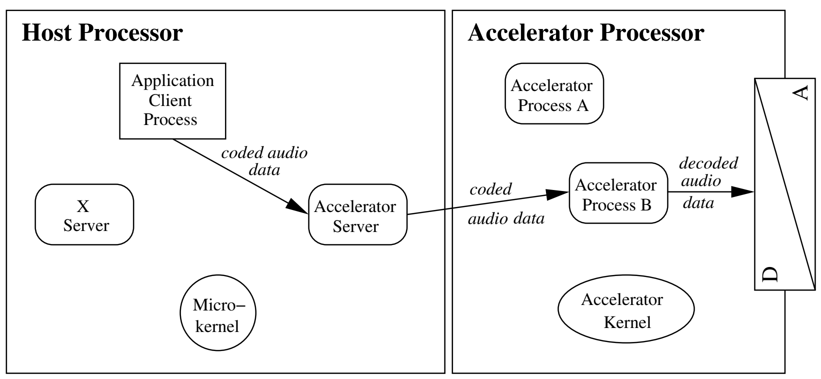


Figure 3.4.: Data Flow in a Basic Interaction Example

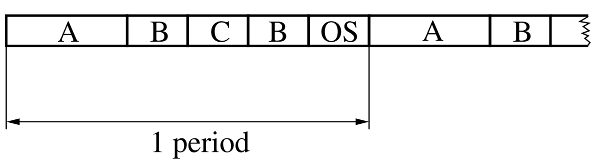


Figure 3.5.: Example Schedule Kernel

4. Implementation Description

Chapter 3 described the principal architecture of a host system exploiting an accelerator subsystem. This chapter presents some relevant aspects of a concrete implementation of such a system. The first section describes data structures, mechanisms and the global operational regime of the accelerator kernel. Section 4.2 focuses on implementational aspects of the host server components. Section 4.3 briefly describes the requirements a DSP application must meet to be executed by the accelerator kernel. The chapter is concluded by a discussion of some design flaws of the accelerator hardware which influenced the implementation in section 4.4.

4.1. SPROS Kernel

We chose the name SPROS for the accelerator kernel, the acronym stands for *S*ignal *PR*ocessing *O*perating *S*ystem. It is an implementation of the architectural concepts described in section 3.3.2 for the DSP TMS320C4x.

4.1.1. Data Structures

The data structure describing an individual DSP process within the SPROS kernel is the `TaskArray`[1]. One entry is associated with one distinctive DSP process and consists of the following parameters of one word each:

- the contents of the process' stack pointer SP,
- the value to be loaded into the timer according to the process execution time per activation t_e,
- the process state,
- the execution entry point of the process (start address),
- the address of the process input buffer (if any),
- the address of the process output buffer (if any).

[1]The name of this data structure is somewhat misleading, because it describes *processes* but implementation was almost finished when the contradiction was discovered.

The TaskArray has a maximum size of 384 words thus limiting the number of different DSP processes present in the system to 64. The timer value depends on the actual DSP hardware configuration and is obtained for the utilized TMS320C44 by

$$timer - value = \frac{f_c}{4} \cdot t_e ,$$

where f_c denotes the DSP's clock frequency and t_e denotes the execution time of the process per activation.

The state of a process can be one of the following:

TASK_RUN: the process is executing,

TASK_INIT: the process has been downloaded, but not executed yet,

TASK_OS: the process is the SPROS kernel itself,

FREE: the entry is currently unused.

The *schedule* is simply a sequence of indices into the TaskArray. Its size is variable but limited to 256 entries (this is no contradiction to the limit of 64 *different* processes within the system, because it is legal to activate a process several times within one schedule period). Therefore the numbers contained in the schedule are in the range $\{0 \ldots 63\}$. The succession of processes in the schedule is repeated continuously for the system's lifetime.

4.1.2. Context Switch

The SPROS kernel utilizes a timer-activated context switch. Whenever a timer interrupt occurs, the associated interrupt service routine (ISR), which is a part of SPROS, is activated. The following mechanism is executed:

1. The registers are saved within the context of the process. This is simply a sequence of push instructions.
2. Communication interrupts (to/from host) are disabled.
3. The actual stack pointer (SP) register is saved within the TaskArray (cf. section 4.1.1).
4. The pointer into the schedule is incremented (and wrapped around, if necessary).
5. The new index into the TaskArray is determined.
6. The state of the process to be activated next is obtained from TaskArray. Depending on this state, one of the following alternatives is chosen:

   TASK_RUN: The contents of the process stack pointer is restored from TaskArray.

TASK_OS: Communication interrupts are enabled and the contents of the SPROS stack pointer is restored from the TaskArray.

INIT_TASK: The process stack pointer is set to init_stack_start which is the address of a dummy stack containing initial values for all registers excluding the program counter PC. Then, the process entry point is read from TaskArray and transferred into the register set template (as initial contents of the program counter register). Finally, the process state is set to TASK_RUN.

7. The timer is stopped, the new timer value is transferred from the TaskArray to a timer status register and the timer is restarted.

8. The registers of the process to be activated are restored by a sequence of pop instructions.

The return-from-interrupt instruction retiud transfers control to the process activated by the just described mechanism.

The calculation of the schedule is described in section 4.2.3.

4.1.3. Operational Regime

The DSP kernel (excluding the context switch described in section 4.1.2) is scheduled as a “normal” DSP process. Upon every activation, it follows the regime depicted in figure 4.1. Basically, the SPROS kernel consists of three consecutive data transfer parts with decreasing priority. Firstly, the kernel checks, whether the host requires the download of a new schedule. If this is the case, this download is performed. The amount of data to transmit is small and of variable length, therefore mailbox communication is employed[2]. Next, the kernel scans the TaskArray, whether any process or the host demands data transmission. If so, this transmission is performed using busmaster DMA. Lastly, the kernel checks, whether the host requires the download of a new process. If this is the case, the download is initiated. It is performed by mailbox communication for two reasons:

- The length of a code download is variable.
- The transmission shall be interruptible anytime.

During this last kernel part, the kernel's deadline may elapse, depending on the amount of code to be transmitted. If this happens, the transmission status is saved and the transmission is resumed within the next activation cycle of the kernel. That means, code download can, depending on the amount of previously transferred data (schedule, process data), span several SPROS activation cycles, and more general, the duration of a code download can not be guaranteed (download will terminate, though).

If the kernel's deadline is still not reached, the kernel enters an infinite wait loop for the expiration of its deadline.

[2]The minimum size of a busmaster DMA transfer is 512 words, but the schedule has a maximum length of 256 words, so for our hardware environment we were forced to use mailbox communication here.

4.1.4. System Initialization

On system boot time (and on a reset) the following functions are executed on the DSP:

- A stack of 200 words is allocated for SPROS,
- The DSP timer is initialized and stated for the first time according to t_e of SPROS,
- the PCI controller of the DSP board is initialized and,
- a mailbox is configured as interrupt source.

In the beginning, the SPROS kernel is the only entry in the `TaskArray` (at position 0). The schedule contains only a zero and is one element long. Therefore, SPROS is activated continuously until the first accelerator process is transferred and included into the schedule.

4.1.5. Memory Layout

SPROS spans the memory regions detailed in table 4.1. The section `.vecs` contains the

Start Address	Length [$words$]	Name
0x00	100	`.vecs`
0x100	300	`.data`
0x400	200	`.talker`
0x600	200	`.stk`
0x800	800	`.code`

Table 4.1.: SPROS Memory Layout

(overlayed) interrupt and trap vector tables. The `.data` section stores all relevant data structures and variables (e.g., the `TaskArray`, the schedule etc.). The `.talker` section contains the firmware boot and communication code, which is transferred at system's boot time from an EEPROM. This code is used for debugging and post-mortem analysis. The SPROS stack is located in section `.stk`, and finally, the SPROS code is stored in section `.code`. In summary, the whole kernel code comprises approximately 2 KWords of memory.

4.2. Host Server Architecture

This section details some of the relevant implementation aspects of the host server. It basically consists of five parts: the frontdoor, a set of evaluators, the admission control, the installer and the transceiver. Conceptual aspects have already been discussed in section 3.3.1.

4.2.1. Frontdoor

The main functionality of the *frontdoor* is a name service as described in section 3.3.1.1. For reasons of simplicity, the information about the installed services is stored in a fixed-length *service table*. Services are identified by character strings, for instance, "`mpeg1_dec`" represents decoding of layer one MPEG audio streams and "`sample_pb`" stands for playback of digitized audio data. The identificator of the registering L4 thread is also stored.

After initialization, the frontdoor enters a continuous loop of waiting for an IPC from a client, performing one of four possible services and sending an answer message back to the client. The services are:

`names_register`: The server makes a new entry in the service table, provided the service has not been registered already and the maximum number of services is not exceeded.

`names_unregister`: If the service to be unregistered is in the service table and the unregistering thread is the same as the registering thread, the entry is removed.

`names_query_name`: The identificator of the registering thread is sent back to the client, provided the queried service has been registered.

`names_query_id`: The names of all services registered by a certain thread (identified by its identificator) are sent back to the client. If the queried thread has not registered any service, an empty string is sent back.

4.2.2. Evaluators

The current implementation provides only rudimentary evaluator functions. The processing requirements of the provided services are hard-coded into the server. Table 4.2 lists some evaluator constants of our MPEG decoder implementation. The stream parameters consist of layer number, sampling frequency and target bitrate, respectively.

Stream Parameters	t_p $[ms]$	t_p $[cycles]$	t_e $[ms]$
I, 44.1 kHz, 128 KB/s	3.25	81250	8.71
II, 44.1 kHz, 96 KB/s	8.35	208750	26.12
II, 44.1 kHz, 128 KB/s	9.53	238250	26.12
II, 44.1 kHz, 192 KB/s	9.68	242000	26.12

Table 4.2.: Processing Requirements for MPEG Decoder

To generate these parameters, a set of different streams (including real-word and synthetic signals) with corresponding target bit rate, sampling frequency and layer number have been decoded using the DSP implementation [BSH98]. The decoding process was analyzed on a frame-per-frame basis by a very accurate DSP simulator. The obtained maximum frame decoding times are denoted by t_e. The period t_p of the MPEG decoder is determined by

$$t_p = \frac{f_{smp}}{framelength} ,$$

where f_{smp} denotes the sampling frequency in Hz and $framelength$ is the length of an MPEG audio frame in $words$.

An improved server version should provide

- a service to dynamically adapt this information according to the installed services (onto registering of these services),
- a way of exploiting the different processing requirements on a stream-per-stream basis[3], for instance by employing stream descriptions.

4.2.3. Admission Control

The *admission control* host server component is responsible for the calculation of a schedule. The algorithm is a simplified version of the Rialto scheduler [JRR97]. In the following, a brief implementation description of the scheduling algorithm is given.

As already stated in section 3.3.1.3, a binary tree stores a quantized representation of DSP execution time. This tree, the *availability tree*, is constructed on system initialization and updated every time a new accelerator process is scheduled.

A fundamental parameter of the system is the so-called *base period* which may be chosen arbitrarily. The base period is the minimum of the periods t_p of all accelerator processes taken into consideration for execution. DSP execution time is distributed at the base period's granularity.

The nodes of the availability tree represent a recurring execution interval within the DSP schedule. Each node has three attributes:

- its period t_p, which is a 2^n-multiple of the base period t_b and corresponds to its depth within the tree.
- its associated amount of DSP execution time still available for DSP processes t_{free}. Initially, this amount is the same for all nodes and equals the base period ($t_{free} = t_b$).
- a list of DSP processes which are associated with the node. Initially, this list is empty.

The depth of the tree is determined by the base period t_b and the maximum possible period of a DSP process:

$$depth = \left\lfloor \log_2 \left(\frac{max(t_p)}{t_b} \right) \right\rfloor$$

[3] We demonstrated the feasibility of this approach in [BSH98].

If a new process is to be introduced into the system, its period is reduced to the largest 2^n-multiple of the base period which is smaller than the process' original period. Then, its execution time t_e is adapted so that the process' execution time requirement is still fulfilled.

Example: *Consider a scheduler with a base period of one millisecond ($t_b = 1$ ms). A DSP process π to be installed requires $t_e = 6$ ms execution time every 15 ms. The largest 2^n-multiple of t_b, which is smaller than 15 ms, is 8 ms ($\leadsto t_{p,\pi} = 8$ ms). The execution time of π is changed to $t_{e,\pi} = 3$ ms.*

Next, the availability tree is traversed in search of a node with the appropriate period and sufficient free processing time. If such a node exists,

- the process is marked as schedulable under the current DSP workload, associated with the node[4] and t_{free} of the node is reduced by $t_{e,\pi}$,
- t_{free} of all child nodes is reduced by $t_{e,\pi}$,
- t_{free} of all ancestor nodes is adapted: Every ancestor node sets its t_{free} to the minimum of t_{free} of both child nodes.

If no node with enough free processing time was found, $t_{e,\pi}$ and $t_{p,\pi}$ are divided by 2 and the process is reiterated. This is performed until a node is found or $t_{p,\pi}$ is smaller than t_b. In the latter case, the process is rejected as unschedulable under the current workload.

The final step of the scheduling algorithm is the generation of the new schedule from the (updated) availability tree: The tree is traversed repeatedly depth-first from the root to the leaf nodes. Every time a branching point is reached which has been visited before, the other alternative is chosen as traversal path. Traversal ends when every node has been visited at least once. The resulting succession of nodes constitutes the new schedule.

Figure 4.2 depicts an availability tree with six processes of different period and execution time. Their activation order within one period of execution would be: BDCG-BE-BDCH-BE-. After that, the schedule repeats.

The scheduling algorithm has two inherent advantages:

- The schedule overhead is bounded by a constant, the depth of the availability tree. It is not a function of the number of installed processes.
- The algorithm minimizes the number of context switches by scheduling activities as close as possible to their desired periodicity.

If a processes is marked as schedulable, the installer part of the host server is notified.

[4] It is possible to associate more than one process to a node.

4.2.4. Installer

The installer maintains a table of processes which are present (but not necessarily executed) on the DSP. If a new accelerator process has been scheduled, it first checks, whether the code of this process has already been transferred (occurring from a past invocation of this particular process). If this is the case, the accelerator process can be started immediately by the transmission of the new schedule.

Otherwise, code download has to be performed before by the service `INSTALL_TASK` using mailbox communication for reasons discussed in section 3.3.2.1. The download protocol is simple: Each transmitted block contains a block length field, a starting address and a variable number of data words. The transmission is considered secure, no attention is paid to data loss or corruption. This is reasonable for PCI transfers, but for less reliable hardware interfaces a more sophisticated approach using CRC checks could prove useful.

Process installation is finished by transmission of the six-word parameter record for the `TaskArray` entry as described in section 4.1.1 (the record is preceded by the index into `TaskArray`).

Next, the new schedule, calculated by the admission control has to be transferred to the accelerator. As soon as the kernel time slot elapses, the newly installed process is executed (provided, there is at least one invocation of it within the schedule!)

The service "NEW_SCHEDULE" `<lgth>` downloads a new schedule of length `<lgth>` via mailbox 2 to the DSP. The download of a schedule follows a simple protocol:

1. `0x200zz` is written to mailbox 2. The value `zz` is the number of the following words minus one (the maximum length of a schedule is restricted to 256 entries). The set Bit 17 is the identifier of a following schedule, the ISR associated with mailbox 2 reacts accordingly.

2. A sequence of exactly `zz` words is written by the host to mailbox 0, read by the DSP and stored as new schedule.

3. The DSP cycles now between position 0 and `zz` of the schedule.

The new schedule directly overwrites the preceding version. Therefore, when the SPROS slot is left, the new schedule is executed.

Deletion of an accelerator process is done by downloading a new schedule without an invocation of the concerned process. The process code remains physically in memory until the memory region is overwritten due to another process to be installed. This policy reduces the amount of code to be transferred when a small set of DSP services is required frequently.

4.3. Accelerator Process Framework

Due to the missing virtual memory concept for DSPs it is impossible to completely isolate DSP processes from each other and from the SPROS kernel. DSP processes must be trusted not to interfere with the SPROS kernel. Problems arise, when DSP processes originating from unknown or insecure sources are installed. It is even possible to use the accelerator subsystem for spying out or corrupting the host system. Using self-initiated PCI busmaster DMA the accelerator is free to read and write any memory region of the host. Security aspects of the described system are an area of further research.

Up to now, an accelerator process to be integrated into SPROS is required to fulfill the following demands, otherwise it cannot be guaranteed that the accelerator works correctly:

1. Accelerator processes maintain their own private stack. On start of a newly downloaded process, it operates onto a dummy stack allocated by the kernel. Therefore, within the first activation interval, the process must create and switch to its own private stack! If this is omitted, system corruption is possible, as soon as the next DSP process is downloaded and scheduled, because both processes will then operate on the same (dummy) stack. Download usually requires several schedule cycles, therefore it is safe to switch to the private stack at a convenient position of the process initialization code (instead of directly at its start).

2. An accelerator process must not modify DSP resources which are seized by the kernel or influence the global operation scheme of the SPROS kernel. For our hardware platform, these include:

- The memory region occupied by the kernel, the interrupt and trap vector table and the bootloader of the board. Currently, this concerns the local memory region between `0x7fe00000` and `0x7fe00fff`.
- The mailbox number two, which is used for low-latency communication (kernel control).
- Certain global registers, like the Status Register (ST) and the CPU Internal Interrupt Enable Register (IIE). Furthermore, the general-purpose registers R11 and AR7, which are reserved for asynchronous interrupt processing within the kernel, may not be used by accelerator processes.
- Accelerator processes are not allowed to establish private interrupt service routines or otherwise interfere with the interrupt system of the kernel.
- The DSP's internal periphery and associated registers: the four serial communication ports, the two timers (one of them is used for the context switch), the DMA coprocessor and the registers controlling the memory interface.
- The JTAG (Joint Test Action Group) debugging interface of the DSP (which is undocumented, anyway).

3. Every process may maintain up to two input and two output buffers of fixed size and at a predefined memory location. If a buffer is due to transfer (an input buffer has been emptied or an output buffer has been filled by the process), a dedicated trap must be issued by the process. For a few cycles, control is transferred to the kernel which marks the corresponding buffers for subsequent transfer[5]. Then, control returns to the executed process. During a single activation, an accelerator process is allowed to mark a maximum of one buffer to transfer to and from the host. Otherwise, data loss is likely.

In general, SPROS processes are not bound to a certain compiler or development environment. It must be possible to arbitrarily choose the entry address and the memory regions the process is linked to.

4.4. Hardware Peculiarities

Quite a few hardware details of our DSP plug-in board proved difficult to handle. Most of them complicated the code, some even led to performance degradations. We believe that they could be eliminated by a careful board design. The problems concern the interface between the board's PCI controller, the DSP and the host.

To improve future DSP boards suitable for accelerator subsystems and to give rough guidelines for board selection, these design flaws are discussed briefly. The hardware structure of the board is described in some detail in [Inn96], the board's PCI controller is exhaustively described in [App96].

We already pointed out the importance of PCI busmaster DMA. For some obscure hardware reason, only the board may act as master in a busmaster transfer, that means only the DSP is allowed to initiate a transfer. But, in our system, the controlling instance of the accelerator subsystem is the host. Therefore, it would be desirable that the host may initiate transfers as well. Communication overhead would be reduced. The board's PCI controller supports bus mastering initiated from both sides, only the necessary registers have to be mapped into the DSP's physical address space.

Write transfers from the DSP to the host have to be partitioned into chunks of 512 words (2 KBytes) and transferred separately. This strong contradiction to the busmaster DMA concept (which allows transfers of arbitrary length) seriously degrades transfer performance. Strangely, the opposite transfer direction is not limited in this way.

Although the PCI controller provides four mailboxes which all are able to generate interrupts when being written or read, only mailbox number two is connected to the DSP's interrupt logic (is able to generate DSP interrupts). As a consequence, the remaining three mailboxes are useless for communication, because they had to be polled actively, which is unacceptable.

Only one external interrupt source is connected to the AMCC chip. All incoming and outgoing data (via mailbox two) as well as completed busmaster DMA transfers of both directions share the same interrupt service routine. The interrupt service routine must determine, what source raised the particular interrupt. This considerably increases the routine's complexity and is undesirable. This design flaw was one of the most complicated implementation problems.

During an active PCI DMA bus master transfer, no access to the PCI controller's registers is allowed (their contents is undefined). The DSP must wait until transfer

[5] The transfer of the buffer is performed during the next activation of the kernel.

completion. The completion raises an interrupt, but the associated interrupt service routine is shared by several other interrupt sources. Unfortunately, there is no way to determine the interrupt source without accessing the registers of the PCI controller. In practice this imposed no problems, because DMA transfers are performed in a few cycles, but if another interrupt is raised during the transfer, system corruption is likely (the transfer is aborted prematurely).

DMA is asymmetric. Whereas the transfer from and to the host buffer to the PCI controller runs transparently, the DSP has to write and read the PCI controller FIFO on its own. This seems to be a shortcoming of the used PCI controller, but differs from controllers of other vendors.

The size of a buffer to be transferred with PCI busmaster DMA must be a multiple of the FIFO length (512 words). Data transfers of arbitrary length ($n \cdot 512 + m$ words) have to be divided into a DMA transfer of length $n \cdot 512$ words and m word transfers using mailboxes. However, this is usually not a major concern for multimedia stream transmission.

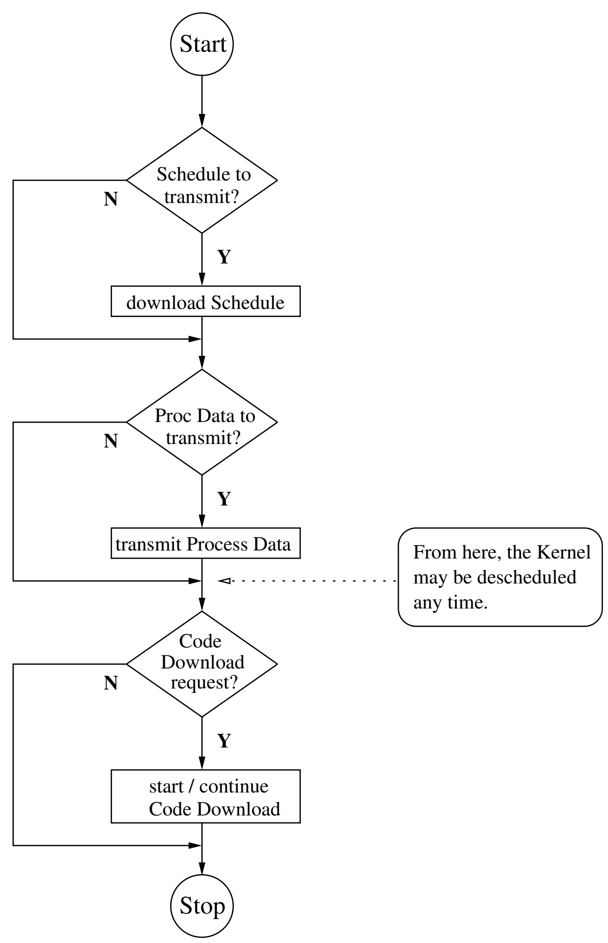


Figure 4.1.: SPROS Operational Regime

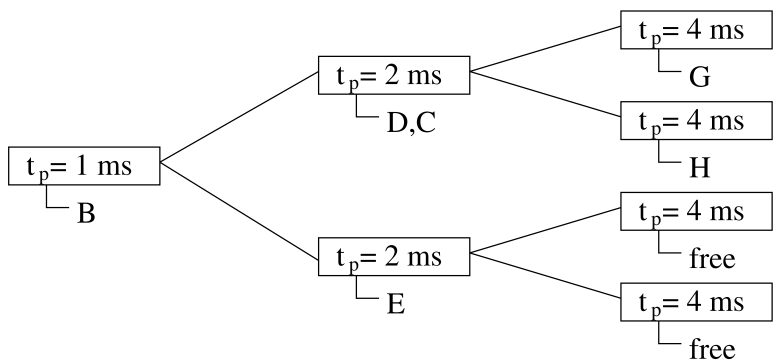


Figure 4.2.: Example Availability Tree

5. Performance

As stated before, the goal of this work is to demonstrate the feasibility of Digital Signal Processors as flexible multimedia accelerators in microkernel environments. To this aim, in the preceding chapters we formulated requirements to be met by DSP-based accelerator systems. Then we discussed aspects of the design and implementation of such a DSP-based accelerator system. Numerous allegations concerning the performance of processors, mechanisms and algorithms have been made. Yet the question remained unanswered up to now, whether the constructed system fulfills the requirements defined earlier.

This chapter fills this gap. It is divided into two parts. In the first part we demonstrate the efficiency of the constructed system. The SPROS DSP kernel is analyzed, evaluated and compared with other DSP kernels. Next, we concentrate on communication performance between host and accelerator subsystem. We describe the performed experiments in detail, present communication performance numbers and evaluate the analyzed communication mechanisms.

The chapter's second part tries to evaluate the work from a user's point of view by trying to demonstrate the benefits a user draws from employing a DSP-based accelerator. First, the discussion focuses on raw DSP processing performance. Regardless of the implications in benchmarking processors described in section 2.1.3.2, we try to estimate the potential processing capabilities of DSPs in comparison with general-purpose processors in respect to multimedia algorithms. We introduce the implemented benchmarks, present obtained performance numbers and try to reason why DSPs render ideal as multimedia accelerators. Additionally, we go into some detail evaluating the impact of MMX onto GPP performance. Second, we analyze two application examples which are implemented and run within the SPROS context of our accelerator prototype, and medium-complexity MPEG audio decoder and an MPEG audio encoder which stresses our current hardware platform to the limit. Again, relevant performance numbers are presented, discussed and compared to implementations requiring no accelerator hardware.

In the final part of this chapter we argue that there is no simple metric to decide, whether or not to execute a particular application on an accelerator or exclusively on the host's CPU. Subsequently, we introduce and discuss informal criteria to assist this decision.

5.1. System Performance

5.1.1. DSP Kernel Parameters

This section presents relevant performance parameters and features of the DSP kernel. As comparison instance we refer to the quasi-industry-standard SPOX kernel as described in section 2.3.2. Table 5.1 compares the key features of both kernels.

SPROS	**SPOX**
preemptive scheduling	cooperative scheduling
static scheduling	static scheduling
support for dynamic changes of process structure	changes of task structure require compilation and DSP reset
timing of system services exactly predictable	timing details of system services not published
certain adaptation of applications necessary	only usable with programs implemented in C
not portable	well portable
minimal functionality	rich set of abstractions and functions (dynamic configurability, device driver abstraction, streams)

Table 5.1.: Comparison of Key Features of SPROS and SPOX

It is obvious that an exact comparison between the two kernels is impossible because of their different architecture and target application area.

The preemptive SPROS kernel ensures that a DSP accelerator process violating its deadline will not interfere with other processes (the process is descheduled regardless whether it has finished execution within its time slot or not) resulting in a better system stability in comparison with a cooperatively scheduling kernel as SPOX.

Both kernels employ static scheduling. The installation and removal of DSP accelerator processes at system runtime is supported in SPROS. In contrast, SPOX's task structure is static at runtime. (All applications and necessary SPOX routines are compiled and linked statically. The resulting binary image is then transferred as a whole to the DSP.) Therefore, a change to the task structure of SPOX requires a compiler run, a DSP reset and a download of the binary.

SPOX is distributed as "black box"; its source code is not publicly available. This prevents an exact timing analysis which is mandatory for guaranteeing real-time services.

Both operating systems require modifications to applications which are to be executed under control of the respective operating system. Additionally, SPOX tasks have to be implemented in C introducing significant overhead (compare for instance [ŽSWS95]).

As a disadvantage, the SPROS kernel is not portable, the minimal set of operating system functions has to be re-implemented for every DSP architecture. This ensures maximum performance. On the other hand, the SPOX kernel is portable. It has been ported to a wide range of processor architectures.

The goal of this work was to demonstrate that for an accelerator kernel a minimum set of operating system functionality is sufficient. The identified essential functions of SPROS comprise:

- process installation and removal,
- process scheduling,
- management of buffer memory,
- host communication,
- management of peripheral devices.

In comparison, SPOX provides a number of higher-level operating system abstractions and functions as semaphores, message queues and device drivers which are not necessary for an accelerator kernel.

Table 5.2 compares the context switch time, the size and the interrupt latency of SPROS with the TMS 320C4x versions of SPOX [Spe97] and Diamond RTOS [3L], another DSP realtime operating system. Further operating systems exist for the TMS 320C4x DSP (e.g., Virtuoso by Eonic Systems or the Precise/MQX kernel), but exact performance numbers are usually not published. It should be noted that the perfor-

Operating System	SPROS	SPOX	Diamond RTOS
Clock Frequency [MHz]	50	50	50
Context Switch Time			
[μs]	4.4	5.0	5.0 ... 7.0
[*instruction cycles*]	80+30=110	125	125 ... 175
Minimum Kernel Size [*words*]	2000	<2000	≈3000
Interrupt Latency [*instruction cycles*]	3	20	N/N
Maximum Number of Processes	64	∞	∞

Table 5.2.: Kernel Parameters: SPROS vs SPOX and Diamond RTOS

mance parameters for SPOX and Diamond RTOS were not measured or analytically estimated but extracted from vendor publications.

The context switch time of SPROS amounts 110 instruction cycles. Because the processor has 40 registers which have to be preserved on a per-process-basis, and a save or restore operation of a register costs one instruction cycle, the theoretical minimum for a context switch is 80 cycles. Hence, the overhead introduced by the mechanism itself amounts only 30 cycles.

The number for the minimum SPOX kernel size is possibly inaccurate, because [Spe97] states "less than 2k words of code" without mentioning space needed for data structures. The code size of SPROS is 1000 words, but additional 1000 words are consumed by stack, data and vector tables which also belong to the kernel (the communication buffers per process were excluded, because they are managed by the kernel but are associated with the accelerator processes). Besides implementing SPROS in assembler language, no special attention was paid to minimize the kernel's size, because for an accelerator subsystem, memory consumption is not a critical system parameter (as it is for embedded systems for example).

The interrupt latency of SPROS for interrupts associated with D/A and A/D converters is reduced to the latency of the processor, three instruction cycles, because they are not deferred at any time as discussed in section 3.3.2.1 (the corresponding interrupt service routine has to well-behave, of course). However, pending interrupts associated with communication are delayed until the next SPROS slot! On the other side, SPOX switches off interrupts for arbitrary periods of time thus increasing its worst-case interrupt latency for all interrupt sources. Interrupt latency numbers for Diamond RTOS could not be obtained.

The maximum number of accelerator processes in SPROS is 64, whereas for SPOX and Diamond RTOS the number is limited only by the processor's resources.

5.1.2. Communication Aspects

Crucial for the acceptance of an accelerator is the performance of the underlying communication mechanism. If it is not possible to transfer data very efficiently between both processors the accelerator concept will fail.

In this section we discuss which mechanisms were available, which were chosen for what type of communication and which proved inappropriate for our purposes. Furthermore, we highlight the importance of some implementation aspects of the communication code.

5.1.2.1. Communication via ISA

During the project's initial phase, an ISA-based DSP plug-in board was the first experimental platform. The actual aim of this experiment was to demonstrate that

- it is possible to relocate certain functionality from the system's CPU to a connected DSP, and
- this concept does not require excessive effort

in a microkernel-based environment. For reasons of simplicity, the educational microkernel VSTa [Val95] was chosen as host operating system (L4 was not yet available). The kernel was complemented by a server responsible for communication to the DSP board. As typical multimedia service to be provided by the DSP, simplified JPEG decoding was selected. An application process executed on the host transferred JPEG data to the server, which forwarded it to the DSP. The DSP waited for JPEG data, decoded it on demand and sent raw image data back to the host. The server copied the received image data into the address space of the application process.

The result was somewhat surprising. In comparison to a JPEG decoder executed exclusively on the host, the performance of the DSP-supported variant was degraded by almost a factor of ten. Subsequently we instrumented the DSP code to immediately send the received data back to the host to analyze the influence of the communication onto the poor performance. Table 5.3 shows the achieved data transfer rate for different message sizes (Test Configuration: Host: Intel Pentium, 60 MHz, 8 MBytes RAM, μKernel VSTa; DSP board: ISA, Motorola 56001, 20 MHz, 96 KWords SRAM). Up to

Message Size $[KBytes]$	Transfer Rate $[KBytes \cdot s^{-1}]$
300	90.4
384	87.4
468	87.3
768	88.4
1200	84.4

Table 5.3.: Transfer Performance of ISA Mailboxes

67% percent of the total DSP processing time was consumed by communication. We concluded that the utilized mailbox communication and ISA bus are at least an order of magnitude too slow for our purposes. Most 1st-generation DSP plug-in boards solely provide mailboxes thus disabling them as multimedia accelerators (though, the DSPs themselves provided sufficient processing power).

The reasons for the very low performance and the almost constant transfer rate of this first prototype were analyzed accurately in [BH96] and [Bau95].

5.1.2.2. Communication via PCI

With the availability of a recent PCI-based DSP board this experiment was reiterated to obtain real-world numbers for the transfer performance of PCI using different mechanisms (mailboxes, PCI busmaster DMA, cf. section 2.2.3). The host's operating system proved unimportant for this estimation, so we migrated from VSTa to Linux, considerably simplifying the implementation of the testbed. We measured one-way communication in the following way. A process executed on the host sent messages of different size using either mailboxes or busmaster DMA to the DSP. The DSP confirmed the receipt. The host measured the time from sending the first word of the message until receipt of the DSP's confirmation. Overhead introduced by signalling and measurement was eliminated by subtracting the transfer time of an empty message.

Due to another lesson learned from the experiments described in section 5.1.2.1, special attention was paid to the optimization of the communication code, the test bed was implemented in hand-optimized assembler code on both processors. Table 5.4 presents the obtained performance numbers using the mailbox mechanism (Test

Configuration: Intel Pentium Pro, 200 MHz, 64 MBytes RAM, Linux Kernel 2.0.34; DSP board: PCI, TMS320C44, 50 MHz, 256 KWords SRAM; Mailbox: 32 Bits).

Message Size [$Bytes$]	Transfer Rate [$KBytes \cdot s^{-1}$]
128	1502.4
512	1544.6
4096	1593.6
32768	1588.9

Table 5.4.: Transfer Performance of PCI Mailboxes

Although the performance was improved by more than an order of magnitude compared to the preceding experiment, it is clearly not sufficient for multimedia stream transmission. The achievable maximum transfer rate is dictated by the expensive sequential signalling mechanism in conjunction with the necessary word-wise transfer. The bare signalling mechanism without data transfer required only a time of 160ns.

We can conclude, that mailboxes are too inefficient for the transmission of multimedia streams regardless of the underlying bus protocol and the degree of optimization applied to the communication code. On the other hand, PCI mailboxes are ideal for low-latency communication with small amount of data, as it is necessary for exchange of control information between the controlling instance on the host and the DSP operating system, provided a carefully optimized implementation.

Many 2nd generation DSP boards provide another communication mechanism, transfer using Direct Memory Access (DMA), which requires little or no CPU rendering it ideal for Multimedia stream transfer.

The next experiment addressed the characteristics of busmaster DMA based on PCI [BH97]. Implementing busmaster functionality proved to be complex on the DSP (cf. section 4.4 for a short description of peculiarities of our hardware environment), we first relied on a high-level language (HLL) communication library provided by the board's manufacturer. In parallel, we developed our own communication code from scratch and applied a careful optimization process to it.

Then, the same measurements as described for mailboxes were performed. The message size could not be kept identically to the mailbox experiment, because, due to hardware reasons, it had to be a multiple of 2 KBytes.

Due to the still unsatisfying results (cf. table 5.5), time-critical functions in the communications library were profiled. We found out that the DSP spent 25% of the overall communication time within the interrupt service routine responsible for data transmission. These functions were subsequently substituted by hand-optimized assembler code employing the following techniques:

- relocation of loop-invariant code out of loops,
- utilization of hardware looping,
- elimination of pipeline stalls by careful instruction scheduling,

- Harvard Architecture exploitation (parallelism of transfer and arithmetic operation)

As a result, we were able to cut down the length of the innermost data transfer loop from 15 to one instruction cycles!

The performance improvement was dramatic, the optimization led to almost ten times faster communication! Table 5.5 presents the combined results of the measured communication performance of PCI busmaster DMA for the vendor's original code and for our own optimized communication library (Test Configuration: Host Intel Pentium Pro, 200 MHz, 64 MBytes RAM, Linux; DSP board: PCI, TMS320C44, 50 MHz, 256 KWords SRAM, AMCC S5933 PCI Controller).

Message Size $[KBytes]$	Transfer Rate $[KBytes \cdot s^{-1}]$	
	Vendor Code	Optimized Code
2	2899	18182
4	3053	25000
8	3150	28571
16	3200	31373
32	3219	32653
64	3232	33508
128	3239	-[1]

Table 5.5.: Transfer Performance of PCI Busmaster DMA

The achieved transfer rates demonstrate the potential performance of the PCI bus. We observed a comparatively high per-transaction overhead, indicated by substantially slower transfer rates for short messages. The results emphasize the necessity of assembler language and manual optimization for time-critical DSP code. The "bare" signaling mechanism requires 15.5 μs (approximately 390 DSP instruction cycles) mainly consumed by the necessary PCI controller programming.

We can conclude that PCI busmaster DMA is a suitable communication mechanism for multimedia stream transfer. Although, it requires a very efficient implementation. For short size messages requiring low latency, PCI busmaster DMA must be complemented by another communication mechanism.

Due to hardware unavailability, we were not able to measure communication performance using shared memory. The published performance evaluations indicate the suitability of the communication mechanism for multimedia stream transfer [Köh98].

Diagram 5.1 summarizes the achieved communication performance of PCI using different mechanisms.

[1] Our optimized communication library is only able to handle a maximum message size of 64 KBytes.

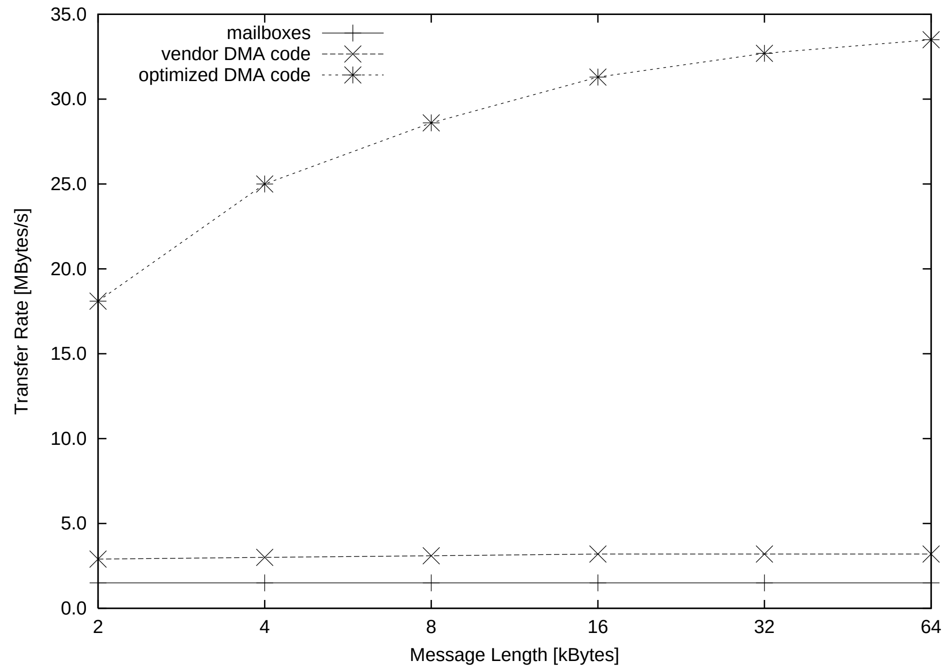


Figure 5.1.: Raw PCI Communication Performance

5.2. DSP Performance

The main topic of this work is the usage of DSPs as accelerators. However, up to now, the question remained unanswered, if a DSP at all is able to execute a given algorithm faster than a general purpose processor. For almost a decade this has been a thoroughly researched subject. For a brief review of important publications, we refer to section 2.1.3.2. Because of the rapidly evolving processor technology, it is necessary to continually evaluate the performance of DSPs in comparison to general-purpose processors and to estimate their respective processing capabilities in the future.

Within the context of this work, a certain amount of processor benchmarking has been carried out. Selected results are presented in the following.

5.2.1. Elementary Benchmarks

The idea itself to use DSPs as accelerators grew during an introductory part of the project. Initially, we wanted to find out, how a DSP performed in comparison to a general-purpose processor depending on the executed code.

We analyzed an off-the-shelf fixed-point DSP, the DSP Motorola 56001 [Mot86] and the widely-used Intel Pentium processor [Int97c]. We implemented a variety of simple problems in assembler language and included a careful manual optimization of the generated code [BH96]. The testbed included problems such as:

- string processing (comparison, copying),

- elementary numeric operations (vector addition, vector matrix multiplication),
- signal-processing algorithms (filters).

The tests were performed with variable amount of data (string, vector and matrix size, filter length). Diagram 5.2 illustrates some of the obtained results (Test Configuration: GPP: Intel Pentium, 60 MHz, 8 MBytes RAM; DSP: Motorola 56001, 60 MHz[2], 96 KWords SRAM).

The string comparison demonstrated the weakness of DSP architectures: code containing many conditional branches is executed slower as on a general-purpose processor. Vector addition and vector-matrix multiplications show a significant performance advantage for DSPs, because multiplications and additions can be performed very efficiently. The highest speedup for DSPs can be achieved by signal-processing algorithms, as the FIR filter example shows.

The benchmark's result can be summarized as follows:

1. DSPs are able to execute certain types of applications faster than general-purpose processors.
2. The implementation has to be optimally tailored to the DSP's individual architecture. As an example, the DSP code should exploit hardware looping and parallelism, minimize the number of conditional branches and use the native number format of the chip. General-purpose processors are not that sensible to code optimization.

One of the evaluated algorithms was an image processing Sobel filter. It calculates the equation

$$y = \sum_{x=1}^{N} n(x) \cdot m(x) + \sum_{x=1}^{N} n(x) + \sum_{x=1}^{N} m(x)$$

for vectors n, m of arbitrary length N. We implemented and evaluated the code for two different MMX processors (Intel Pentium MMX/166, Intel Pentium II/300) and the DSP TMS320C4x. Table 5.6 presents the execution time for processing a vector of 8192 elements. Additionally we include benchmark results for the DSP TMS320C6201 published in [CHE97a] to demonstrate the capabilities of recent DSPs. The general-

Processor	Clock Frequency $[MHz]$	Execution Time $[\mu s]$
Pentium MMX	166	231.5
TMS320C44	50	491.8
Pentium II	300	128.1
TMS320C6201	200	81.9

Table 5.6.: Sobel Filter Performance of different Processors

purpose processors outperformed our current DSP for two reasons:

[2] The DSP results were obtained from a 20 MHz processor and scaled accordingly. This was justifiable within the experimental environment.

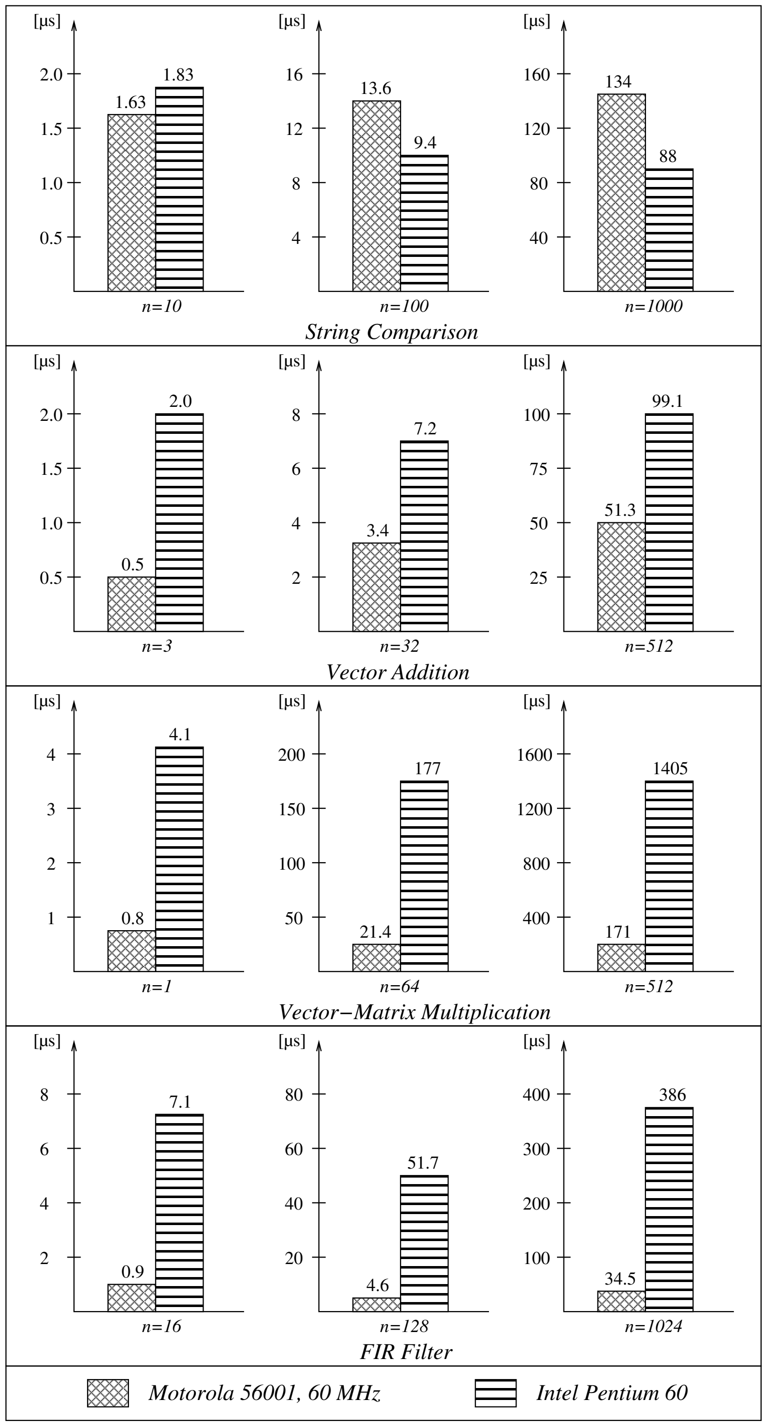


Figure 5.2.: Elementary Benchmarks on GPP Intel Pentium 60 and DSP Motorola 56001/60

1. The chip design of the DSP is six years older (the TMS320C4x has been introduced in 1990, whereas MMX emerged 1996).
2. The DSP is optimized for processing floating-point numbers, whereas MMX can only operate on fixed-point data. The filter was implemented using fixed-point-data.

Nevertheless, it seems that actual DSPs maintain the performance advantage of DSPs over general-purpose processors on signal-processing-style algorithms.

Furthermore, we wanted to find out, how MMX performed in comparison to 'classic' ix86-style code. Diagram 5.3 visualizes the determined instruction cycles per vector element for the Sobel filter for our DSP and an Intel Pentium with and without MMX support, respectively. For the regarded code and processors we can conclude:

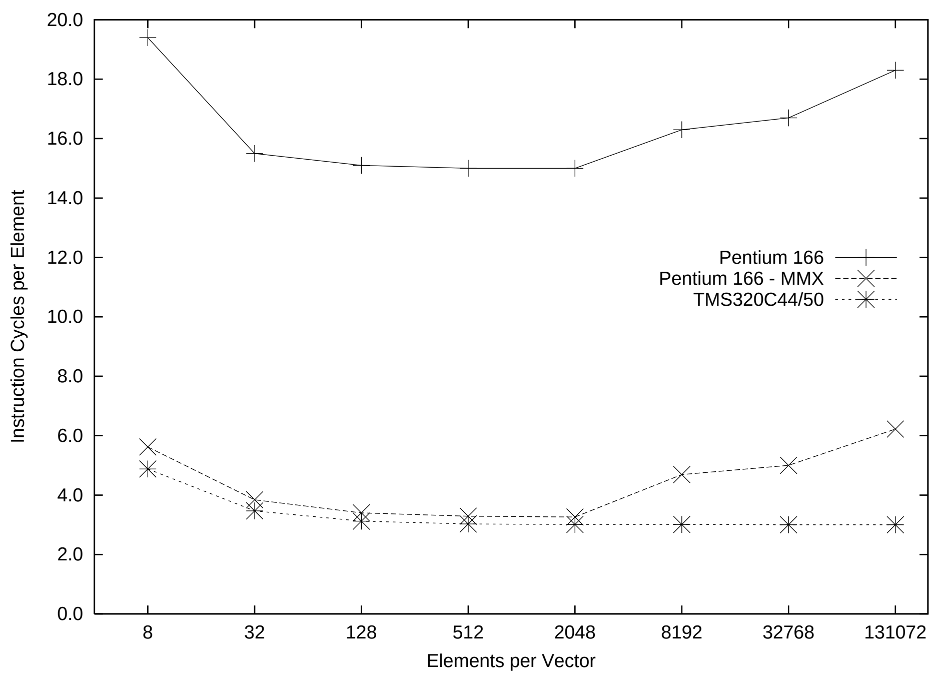


Figure 5.3.: Required Instruction Cycles for Sobel Filter on Different Processors

- MMX reduced the number of required instruction cycles by a factor of four in comparison to non-MMX code.
- For small amounts of data[3], MMX needs approximately the same number of instruction cycles as the DSP.

We can conclude that MMX increases the x86 architecture's performance on signal processing style algorithms processing fixed-point data. Nevertheless, a performance advantage for DSPs persists.

[3]The data set must fit within the processor's first-level cache.

5.2.2. Case Study: Encoding and Decoding MPEG Audio Streams

This section presents relevant performance parameters of an MPEG audio *decoder* as a medium complexity application and a complex fully-featured MPEG audio *encoder*.

5.2.2.1. MPEG Audio Decoder

The objective for this project's part ([Stu97]) was to implement an MPEG audio decoder for layer I and layer II streams which executed in real-time, that is, required less execution time for decoding a frame than is available for playback of a frame. Table 5.7 shows the associated time constraints for different sampling frequencies and the maximum measured frame decoding times of the final DSP implementation.

MPEG Layer	Latest possible Decoder Deadline in ms for			Maximum Frame Decoding Time $[ms]$
	$f_s = 32.0\ kHz$	$f_s = 44.1\ kHz$	$f_s = 48.0\ kHz$	
I	12	8.7	8.0	4.6
II	36	26.1	24.0	13.9

Table 5.7.: Time Constraints and Achieved Performance for MPEG Audio Decoding

A part of the analysis focused on methods for the determination of appropriate deadlines for MPEG stream decoding on the accelerator. The traditional solution is to set the deadline according to the playback parameters of the stream. For instance, for a layer II MPEG audio stream sampled at 44.1 kHz and with a frame length of 1152 words, the deadline for one decoder invocation would be:

$$t_{dl} = \frac{1152\ words}{44100\ words \cdot s^{-1}} = 26.1\ ms$$

The suggested approach would perform a preliminary decoder run of the stream in question instead and determine the maximum necessary frame decoding time. This maximum is then chosen as the real decoder deadline for all subsequent decoding runs of that stream. For instance, by analyzing the frame decoding time of layer II streams of different contents we obtained maxima of approximately $9.1 \ldots 9.6\ ms$. This method ensures that no quality degradation is possible during playback and at the same time saves as much as possible processing time of the DSP. Our experiments showed that it is possible to save approximately 65% of DSP processing time that way. More details together with further experiments addressing influence factors of MPEG audio frame decoding time on a DSP can be found in [BSH98].

5.2.2.2. MPEG Audio Encoder

In another side project of the main work, a complete MPEG audio encoder for layers I and II was implemented for the DSP TMS320C44 [Stu98]. Again, the development goal was to encode streams in real-time regardless of the target bitrate and sampling frequency.

Diagrams 5.4 and 5.5 show the achieved performance for stream encoding (single channel mode, 44.1 kHz sampling rate). As the diagrams reveal, this application demonstrates the performance limits of a single DSP TMS320C44 in dual channel mode. Although the whole program was implemented in assembler language and all kinds of optimizations were performed to the generated code, the encoder violates in some situations its real-time constraints (cf. table 5.7).

Due to the restrictive patent policy of the Fraunhofer Institute it is almost impossible to obtain an open source MPEG audio encoder for the Intel Pentium platform. Therefore we are unable to compare our achieved performance to GPP implementations.

5.2.3. Selection Criteria for Acceleratable Applications

In the preceding sections of this chapter, we demonstrated that a DSP-based accelerator subsystem *can* claim a substantial performance advantage over a 'classic' one-processor solution if certain conditions are fulfilled. However, the development of applications to be executed by the accelerator proved to be very expensive due to the complex software development process and the necessary level of code optimization.

Therefore, a concise analysis which applications are worth to be ported to the accelerator and which are to be left executed by the system's CPU is a must.

The analysis of DSP benchmark publications (cf. section 2.1.3.2) as well as our own benchmark research did not lead to a simple metric, whether or not an application should be executed by the DSP or by the CPU. Even assuming an optimum implementation of the algorithm, there are still too many influence factors to consider.

The factors determining the performance of a particular algorithm implementation of a given algorithm include

- the processor's architecture, especially its degree of possible parallelism,
- the clock frequency of the processor,
- the level of code optimization,
- the amount of and access pattern to data to be processed.

As for any other processor it is impossible to generate execution timing information of arbitrary complex algorithms by rules of thumb. We conclude that the evaluation, whether a given application can be accelerated by a DSP, requires at least a prototypic DSP implementation and accurate performance numbers of the algorithm executed by the CPU alone.

It is not possible to evaluate a DSP and a CPU implementation by comparing isolated technical parameters. For example, the maximum clock frequency of common GPPs is approximately two to three times higher than that of typical DSPs. However, this does not result in an overall performance advantage of GPPs, because DSPs usually perform more micro-operations within one clock cycle than their general-purpose counterparts.

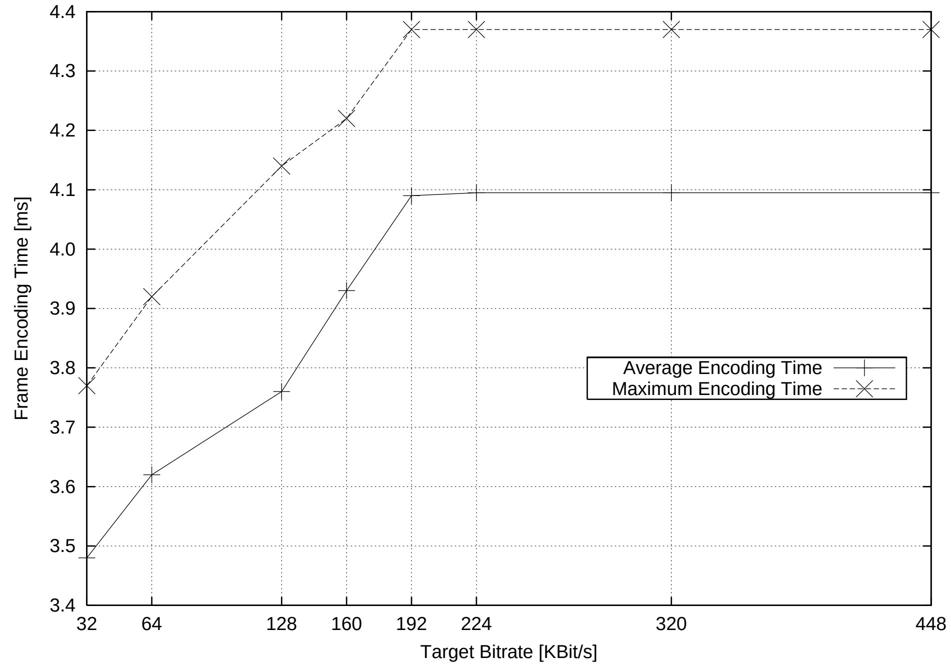


Figure 5.4.: Frame encoding time for layer I MPEG audio streams of different target bitrates

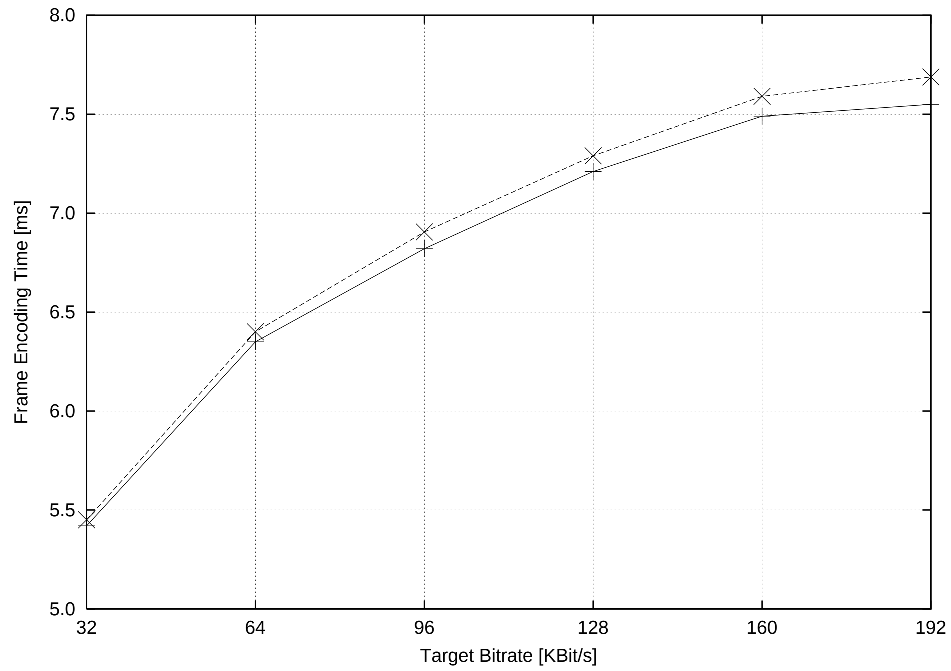
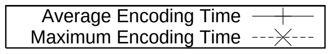


Figure 5.5.: Frame encoding time for layer II MPEG audio streams of different target bitrates



Throughout the project, certain informal characteristics of algorithms particularly suitable or unsuitable for DSPs were identified and gathered. Table 5.8 summarizes key differences between DSP-suitable and GPP-suitable applications.

DSP-suitable	**GPP-suitable**
◦ regular patterns in data indexing	◦ unpredictable data access patterns
◦ data and algorithms partitionable into small blocks	◦ problems not necessarily partitionable
◦ loops with known numbers of iterations, few conditional branches	◦ many data-dependent branches
◦ huge share of operations are multiplications and additions	◦ all types of operations necessary
◦ ratio of operations to be performed to data amount is high (compute-bound execution time)	◦ low operations / data amount ratio (transfer-bound execution time)
◦ volatile data set (e.g., streams)	◦ data set non-volatile
◦ no user interaction	◦ user interaction required or possible
◦ individual data adheres to fixed length or constant format	◦ variable format length or amount of data
◦ few or standardized data conversions	◦ complex data conversion
◦ arithmetic operations can be accompanied by load/store instructions[4]	

Table 5.8.: Differences between Applications suitable for DSPs and GPPs

If an application promises acceleration by means of a DSP, the next point to be considered is an optimum implementation. Apart from implementing the algorithm in maximum conformance to table 5.8, the following conditions must be fulfilled to ensure maximum performance.

- The implementation exploits the special DSP features[5].
- The implementation maps the algorithm onto the DSP architecture such that in every instruction cycle a maximum of the DSP's units are busy.
- At least time-critical parts of the implementation are coded and hand-optimized in assembler language.

[4] provided that the actual DSP architecture allows parallel data transfer
[5] Due to the diversity of DSP architectures, detailing this note is beyond the scope of this work.

6. Conclusion

This chapter briefly recalls main results of this work. Besides, we give a short outlook on further research and improvements.

The principal result of this work is the derivation of a architecture that allows to move time-critical multimedia processes from the system's CPU to an accelerator subsystem. As was thoroughly discussed throughout the work, Digital Signal Processors (DSPs) proved ideal as accelerator processors for this kind of processes. The software architecture ensures that realtime guarantees, once given, are sustained.

The work revealed the importance of the hardware interface between host processor and accelerator. As we pointed out, certain performance requirements, concerning data transfer rate and latency must be fulfilled. Peripheral Component Interconnect (PCI) proved acceptable as communication mechanism.

The DSP's performance capabilities are exploited to a maximum by executing a minimal kernel on the DSP. This kernel provides context switching of accelerator tasks based on a previously generated schedule, services for installing and removing DSP processes and communication functions between host and DSP. The size of this kernel can be kept as low as a few kilobytes.

The controlling instance of the accelerator subsystem is executed on the host. Because the focus of this work is on microkernel-based host operating system structures, it is implemented as a user-level microkernel server. This server performs interaction with user tasks, installation and removal of DSP processes, admission control and schedule generation for the DSP as well as general data transmission between accelerator subsystem and host.

We demonstrated that it is possible to implement complex multimedia algorithms on DSPs by providing a fully-featured MPEG audio decoder and an MPEG audio encoder which both can be executed under real-time conditions within the context of our project. The necessary steps for the integration of a new DSP process are:

1. provisional estimation of performance speedup by accelerator implementation

2. design, implementation and optimization of the DSP code

3. measurement of exact processing requirements (i.e., estimation of scheduling parameters)

4. integration into scheduling database of the host

Once a certain accelerator function is established, our system guarantees its quality of service. The installation of further accelerator processes does not affect processes already executed. In particular, DSP processes are not able to interfere with each other, even when implemented faulty. The realized preemptive context switch contributes to the system's stability.

Up to now, the system is only able to give guarantees to DSP processes with constant execution time and periodicity. Therefore, further research should be conducted to include processes with variable scheduling parameters. For instance, speech coders employ constant activations but varying deadlines according to the complexity of the actual phonemes. Instead of executing such a decoder with its latest deadline (as performed now), there should be a mechanism to adapt to these variable processing requirements. Generally, it seems promising to incorporate more knowledge about the processing requirements of the DSP processes in order to distribute DSP execution time even more efficiently.

At the present time, the system is only capable of executing one non-realtime process in addition to the set of realtime processes (all remaining processing time is donated to this process). In situations where much processing time is available it would be more convenient to allow more than one such process. Ideal would be a solution without limiting the number of non-real-time processes yet trying to distribute the remaining processing time fairly.

Crucial for the acceptance of accelerator systems is the availability of a broad range of accelerator functions to be executed on the DSP. As demonstrated, implementation of DSP software is expensive for at least three reasons:

- At least time-critical parts of the DSP code have to be implemented in assembler language and optimized carefully, because there do not exist any high-level languages to express the exploitation of certain DSP features.
- Debugging a "remote" processor is more difficult than debugging a program executed on the CPU.
- The diversity of existing incompatible DSP architectures requires a huge amount of programming effort, because it is necessary to establish a code base for every relevant DSP architecture.

Therefore, another field of necessary further research is the general improvement of DSP software technology. There is a significant need of tools for implementing, testing and verifying DSP software. Especially, a method for a more exact approximation of the potential acceleration of a particular algorithm is needed.

Due to the prototypic character of the implementation, the constructed system is not suitable for immediate everyday use. Beforehand, careful testing and verifying has to be applied. Many minor implementational details remain to be clarified.

Up to now almost no attention has been paid to security aspects of the presented accelerator architecture. As suggested in section 4.3, it is possible to exploit the accelerator to attack the host system. To distribute DSP accelerator code over insecure media like the Internet, methods have to be developed to prove the trustworthiness of a particular DSP application.

The acceleration potential of the presented DSP-based subsystems is limited by current DSP processor technology. A promising, yet very challenging approach, is the combination of DSPs with reconfigurable units like Field-Programmable Gate Arrays (FPGA). Such modular accelerator systems could easily be adapted to increasing needs of processing power, though their inherent complexity would be much greater than that of DSP-based accelerator systems.

Theses

This section contains the theses of the work in accordance with the regulations concerning doctoral dissertations of the Department of Computer Science at Dresden University of Technology.

The aim of this work is to prove the feasibility of Digital Signal Processors as flexible, powerful and cost-efficient accelerators for a wide variety of tasks currently executed by the host's CPU. The major statements of the work are:

1. DSPs are very efficient microprocessors. They are freely programmable and therefore flexible yet specialized enough to deliver superior performance for a selected range of applications. Additionally, their price-performance ratio renders them ideal for integration into personal computer systems. The DSP should be embedded within a subsystem together with local RAM, Audio Codecs and appropriate digital interfaces.

2. The specialization of DSP architectures requires a careful selection of algorithms to be accelerated. The work gives guidelines for this selection process. Besides other, the processing of uni- and bidirectional multimedia data streams in conjunction with complex encoding and decoding algorithms proved as successful acceleration candidates.

3. The heterogeneity of current DSP architectures requires a careful selection of the concrete processor type. To this aim a large amount of published processor comparisons were analyzed and many new benchmarks with emphasize on multimedia algorithms were devised. The results permit the comparison of DSPs and general-purpose microprocessors with and without DSP-like instruction set extensions and make it possible to identify DSP architectures which are best capable of accelerating multimedia applications.

4. High-level languages like "C" permit to access the acceleration potential of DSPs only to a certain degree due to the processor's irregular architecture. To fully exploit the processing capabilities of DSPs it is essential to program both DSP system software and time-critical application parts in assembly language and apply a careful optimization process to the code.

5. Transfers executed via the PCI bus system and based on bus master DMA are a viable hardware mechanism for data transport between DSP and Host memory.

The bus systems ISA and SCSI do not match the high bandwidth requirements for multimedia stream transfer.

6. Measurements showed that the poor data transfer performance of current DSP integrations results from:

   - limiting hardware capabilities (bus system limitations, limitations in PCI chipsets et cetera),
   - overhead introduced by the operating system running at the host (monolithic kernel, driver architecture),
   - overhead introduced by the operating system of the DSP, if any (unnecessary functionality, inefficient implementation, unqualified architecture).

   The work demonstrates very efficient data transfer mechanisms between DSP and host memory based on a microkernel architecture running on the host and a optimized kernel running on the DSP. Our mechanisms achieve transfer rates of 35 MBytes/s which is approximately ten times faster than the corresponding mechanisms supplied by the hardware vendor.

7. DSP-based accelerators are capable of accelerating several host applications in parallel by incorporating a consequently optimized DSP kernel doing static scheduling of periodic tasks controlled by a host microkernel server which is responsible for communication, security and synchronization of client host processes. As a concrete example, we designed a DSP kernel for the TI TMS320C44 DSP and a host server running under the microkernel L4.

8. To exploit the processing capabilities of a DSP to a maximum it is necessary to determine very exact deadlines even for complex application processes. Strategies for this determination procedure applied to multimedia algorithms and based on a per-stream analysis are demonstrated.

A. Abbreviations

ADC	Analog-to-Digital Converter
AGP	Accelerated Graphics Port
ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
CD	Compact Disc
CPU	Central Processing Unit
DAC	Digital-to-Analog Converter
DEC	Digital Equipment Corporation
DMA	Direct Memory Access
DSP	Digital Signal Processor
DVD	Digital Versatile Disc
EEPROM	Electrically Erasable Programmable Read Only Memory
EISA	Extended Industry Standard Architecture
FFT	Fast Fourier Transform
FIFO	First In First Out
FIR	Finite Impulse Response
FP	Floating Point
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPP	General Purpose Processor
HLL	High-Level Language
HP	Hewlett-Packard Company
IEEE	Institute of Electrical and Electronic Engineers
IIE	Internal Interrupt Enable (register)
IP	Internet Protocol
IPC	Inter Process Communication
ISA	Instruction Set Architecture
ISA	Industry Standard Architecture
ISR	Interrupt Service Routine
ISSE	Internet Streaming SIMD Extension
IVT	Interrupt Vector Table
JPEG	Joint Picture Expert Group
JTAG	Joint Test Action Group
L4	A Microkernel (developed by J. Liedtke [Lie96])
MAC	Multiply-and-Accumulate

MAX	Multimedia Acceleration Extensions (HP)
MCA	Micro Channel Architecture
MDMX	MIPS Digital Media Extension
MIDI	Musical Instruments Digital Interface
MPEG	Motion Picture Expert Group
MVP	Multimedia Video Processor
OS	Operating System
PC	Personal Computer
PC	Program Counter (Register)
PCI	Peripheral Component Interconnect
QoS	Quality of Service
RAM	Random Access Memory
ROM	Read Only Memory
RSA	Rivest, Shamir, Adleman (cryptographic algorithm)
SCSI	Small Computer System Interface
SIMD	Single Instruction Multiple Data
SP/DIF	Sony and Philips Digital Interconnect Format
SRAM	Static Random Access Memory
ST	Status (register)
TI	Texas Instruments
TLB	Translation Lookaside Buffer
TVT	Trap Vector Table
UDP	User Datagram Protocol
USB	Universal Serial Bus
VESA	Video Electronics Standards Association
VIS	Visual Instruction Set (Sun)
VLB	VESA Local Bus
VSTa	Valencia Simple Tasker (a Microkernel)

B. Glossary

The following list defines some frequently occuring notions and clarifies their meaning within the context of this work.

Acceleration. A service provided by the acceleration subsystem to application client processes. Certain processing-intensive functions are executed considerably faster on the accelerator's processor than on the host processor.

Accelerator, Accelerator's Processor. A freely-programmable microprocessor as central part of the accelerator subsystem. For reasons discussed in section 2.1.3 we concentrate on Digital Signal Processors (DSPs) as accelerators.

Accelerator Kernel. A minimum kernel executed on the accelerator's processor to provide simultaneity of actions on the accelerator.

Accelerator Process. A process executed on the accelerator which provides a defined set of accelerator functionality (e.g., a JPEG decoder). It is run by the accelerator kernel.

Accelerator Server. The microkernel server (executed on the host processor) which is responsible for the management of the accelerator subsystem.

Accelerator Subsystem. A component of a standard-architecture PC. It comprises the accelerator's processor, its memory, peripheral devices (e.g., Digital-Analog or Analog-Digital Converters) and the host interface to the host processor.

Application Client Process. An arbitrary process executed in user mode on the host processor on top of the microkernel.

Busmaster DMA. A communication mechanism. It transfers autonomously (i.e., without CPU interference) data blocks of arbitrary length between main memory and a connected device.

Control-oriented Code. Code containing a high percentage of conditional branches.

Digital Signal Processor, DSP. A class of freely-programmable microprocessors optimized for execution of signal processing algorithms. Throughout this work we use the acronym 'DSP' synonymously for the accelerator's processor.

Guaranteed Acceleration. If a certain acceleration request is granted by the accelerator, it guarantees that the requested service is provided without service degradation until end of this service (which is determined by the client process).

Communication Mechanism. The mechanism for data transfer between host processor and accelerator's processor. Examples are mailboxes, DMA, or shared memory. It is independent from the underlying bus system (ISA, PCI) or interface (RS232, IEEE 1394).

Host Interface. The hardware interface between the host processor and the accelerator's processor.

Host Processor. The CPU of the system.

Inter-Processor Communication. Communication between the host processor and the accelerator's processor.

Interrupt Latency. The duration between the generation of an interrupt signal at a processor's interrupt pin and the execution of the first instruction of the associated interrupt service routine.

Local I/O bus. A bus which directly connects peripheral devices to the system's main memory. Examples are ISA or PCI.

Mailbox. A communication mechanism. Mailboxes employ handshaking to transfer data on a per-word basis.Both communicating processors are involved.

Microkernel. The minimum operating system executed on the host processor. It provides only the most basic abstractions for the implementation of complex operating systems. This project focuses on the microkernels L4 [Lie96], [AH98] and its compatible successor Fiasco.

Peripheral Bus. A bus to connect peripheral devices to the computer system. The peripheral bus needs a local I/O bus to access the processor and main memory. Examples are SCSI, USB or IEEE 1394 (FireWire).

Standard PC architecture. A personal computer system based on an Intel-compatible CPU. The system can be expanded by standardized plug-in boards.

Talker. The bootloader of the DSP plug-in board located on a EEPROM. It occupies 200 words of DSP memory starting at `0x7fe00200`. It is required to bootstrap the DSP.

Bibliography

[3L] 3L Limited. Diamond RTOS 'C4x Edition V2.0. Data Sheet, available from `http://www.3L.com/c4x/dtasheet.pdf`.

[Adv97] Advanced Micro Devices, Inc. AMD-K6 MMX Enhanced Processor. Sunnyvale, CA., June 1997. available at `http://www.amd.com`.

[Adv98] Advanced Micro Devices, Inc. 3D NOW! Technology Manual. Sunnyvale, CA., May 1998. available at `http://www.amd.com`.

[AH98] Alan Au and Gernot Heiser. L4 User Manual. Technical report, The University of New South Wales, Sydney, Australia, January 1998. Version 1.0.

[Ame98] American National Standard of Accredited Standards Committee X3. *Information technology - Small Computer System Interface - 2*, April 1998. Working draft X3T9.2 Project 375D, Revision 10L.

[App96] Applied Micro Circuits Corporation. *S5933 PCI Matchmaker Controller Data Book*. AMCC, San Diego, CA, 1996.

[Bau95] Robert Baumgartl. Integration von Signalprozessor-Hardware in ein Echtzeit-Betriebssystem. Master's thesis, Dresden University of Technology, Dresden, November 1995. in German.

[Ber98] Manfred Bertuch. Kino inklusive – Winfast 3D S800: 2D/3D-Grafikkarte mit DVD-Funktionen. *c't*, (10):80–81, May 1998. in German.

[BH96] Robert Baumgartl and Hermann Härtig. On the Integration of DSP Hardware into a Microkernel-based Operating System. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'96)*, pages 867–872, Boston, MA, October 1996.

[BH97] Robert Baumgartl and Hermann Härtig. Efficient Communication Mechanisms for DSP-Based Multimedia Accelerators. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'97)*, pages 459–463, San Diego, CA, September 1997.

[Bie97] Jeff Bier. Introducing the BDTImark: A New Measure of DSP Performance, October 1997. Presentation Slides, available at `http://www.bdti.com`.

[Bla97] Garrick Blalock. The BDTImark: A Measure of DSP Execution Speed. White Paper, available at `http://www.bdti.com`, 1997.

[BLS+97] Jeff Bier, Phil Lapsley, Amit Shoham, Ole Wolff, Harri Hakkarainen, and Garrick Blalock. *Buyer's Guide to DSP Processors*. Berkeley Design Technology, Inc., Berkeley, CA, 1997.

[BSH98] Robert Baumgartl, Kai Stuhlemmer, and Hermann Härtig. Data Dependency in MPEG Audio Frame Decoding Time on a DSP Accelerator. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'98)*, Toronto, September 1998.

[Car96] David Carson. AGP Architectural Overview, May 1996. slides of a presentation, available at `http://agpforum.org/foils.htm`.

[Cen97] Centaur Technologies, Inc. IDT WinChip C6 processor datasheet. Austin, TX, September 1997. available at `http://www.winchip.com`.

[CHE97a] CHEOPS Bildverarbeitungs GmbH & Co. KG. Simulation of program execution of C6x versus desktop-CPUs. available at `http://www.cheops-bv.de/cheops/c6x-cpu.htm`, 1997.

[CHE97b] CHEOPS Bildverarbeitungs GmbH & Co.KG. Simulation of program execution of C6x versus C4x. available at `http://www.cheops-bv.de/cheops/c6x-c4x.htm`, 1997.

[Chr97] Chromatic Research, Inc. Mpact R Media Processor. Preliminary Data Sheet, 1997.

[Cyr97a] Cyrix Corp. Cyrix 6x68MX Processor. Richardson, TX, July 1997. available at `http://www.cyrix.com`.

[Cyr97b] Cyrix Corp. Cyrix Extended MMX Instruction Set. Cyrix Application Note 108, available at `http://www.cyrix.com`, October 1997.

[DBJ+98] M. Dolle, C. Baumhof, S. Jhand, W. Lehner, O. Müller, and J. Schulte. Hyperstone E1-32XP - New Horizons for Low-Cost Multimedia Applications. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'98)*, pages 1486–1490, Toronto, September 1998.

[Dig96] Digital Equipment Corp. Alpha Architecture Handbook. Maynard, MA., 1996.

[För98] Frank Förster. Firewire-Prototyping. *Design&Elektronik*, pages 66–69, April 1998.

[Ful98] Sam Fuller. Motorola's AltiVec Technology. White Paper, available at `http://mot-sps.com/sps/General/chips-nav.html`, 1998.

[GS96] Jodi Guilbault and Peter Stevens. The Media Processor. White Paper, available at `http://www.trimedia.philips.com/`, October 1996.

[Hal97] Tom R. Halfhill. Beyond MMX. *Byte*, 22(12):87–92, December 1997.

[Hit98] Hitachi Semiconductor Inc. SH-DSP Benchmarking. Application note, Revision 2.4, January 1998.

[Inn96] Innovative Integration, Inc. *PCI44 Hardware Manual*. Westlake Village, CA, first edition, 1996.

[Int97a] Intel Corp. Intel Platforms For Visual Computing. White Paper, available at `http://www.intel.com/intel/march24/techinfo.htm`, March 1997.

[Int97b] Intel, Inc. Intel Architecture Optimization Manual, 1997. available at `http://www.intel.com`.

[Int97c] Intel, Inc. Intel Architecture Software Developer's Manual, 1997. available at `http://www.intel.com`.

[Int97d] Intel, Inc. Pentium Processor Family Developer's Manual, 1997. available at `http://www.intel.com`.

[Int98] Intel Corp. *Accelerated Graphics Port Interface Specification*, May 1998. Revision 2.0.

[JRR97] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP-16)*, pages 198–211, Saint-Malo, 1997.

[Kea96] Liz Keate. Real-Time Operating System Simplifies DSP Multiprocessor Application Development. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'96)*, pages 1265–1271, Boston, MA, October 1996.

[KFG+92] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. The Programmer's View on MARS. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 223–226, Phoenix, AZ, December 1992.

[KFG+93] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. Real-Time System Development: The Programming Model of MARS. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pages 190–199, Kawasaki, Japan, March 1993.

[Kil96] Earl Killian. MIPS Extension for Digital Media, 1996. Presentation, available at `http://www.mips.com`.

[KKS94] Daniel I. Katcher, Kevin A. Kettler, and Jay K. Strosnider. Real-Time Operating Systems for Multimedia Processing. In *Proceedings of the Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.

[KKS95] Kevin A. Kettler, Daniel I. Katcher, and Jay K. Strosnider. A Modeling Methodology for Real-Time/Multimedia Operating Systems. *IEEE Real-Time Technology and Applications Symposium*, 1995.

[Kna95] Steven K. Knapp. Using Programmable Logic to Accelerate DSP Functions. Technical report, Xilinx, Inc., 1995.

[Köh98] Steffen Köhler. Untersuchungen zum Einsatz von DSPs in betriebssystemnaher Umgebung von PCs. Master's thesis, Dresden University of Technology, January 1998. in German.

[KP99] Jagannath Keshava and Vladimir Pentkovski. Pentium III Processor Implementation Tradeoffs. Technical report, Intel, Inc., May 1999.

[KW95] Adam J. Kunzman and Alan T. Wetzel. 1394 High Performance Serial Bus: The Digital Interface for ATV. *IEEE Transactions on Consumer Electronics*, 14(13):893–900, August 1995.

[Lap94] Phil Lapsley. DSP Chips enable PC Multimedia. *Microprocessor Report*, August 1994.

[Lap95] Phil Lapsley. NSP shows Promise on Pentium, PowerPC. *Microprocessor Report*, May 1995.

[LBSL97] Phil Lapsley, Jeff Bier, Amit Shaham, and Edward A. Lee. *DSP Processor Fundamentals*. IEEE Press, New York, 1997.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC Performance (Still The Foundation For Extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, Chatham (Cape Code), Massachusetts, May 1997.

[LH96] Ruby Lee and Jerry Huck. 64-bit and Multimedia Extensions in the PA-RISC 2.0 Architecture. Technical White Paper, available at `http://www.hp.com`, 1996.

[Lie95] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, Colorado, December 1995.

[Lie96] Jochen Liedtke. L4 Reference Manual. Technical report, German National Research Center for Computer Science (GMD), May 1996. Version 2.0.

[Lou97] Loughborough Sound Images plc. Evaluation of the Performance of the C6201 Processor & Compiler. White Paper, Document Release 1.1, available at `http://www.lsi-dsp.co.uk`, 1997.

[Mar96] John Marsh. An Introduction to the TM-66 swiFFT DSP Chip. In *Proceedings of the 1996 DSP^x Conference*, San Jose, CA, March 1996.

[Mar98] Daniel Martin. Unique DSP Architecture blends into the new Siemens 32 Bit μC/DSP TriCore Family. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'98)*, pages 1508–1512, Toronto, September 1998.

[MFH98] Rodney Van Meter, Gregory G. Finn, and Steve Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[MIP97] MIPS Technologies, Inc. MIPS Extension for Digital Media with 3D. available at `http://www.mips.com`, March 1997.

[MM96] John Marsh and John McCaskill. The TM-66 swiFFT DSP Chip – Architecture, Algorithms and Applications. In *Proceedings of the SPIE AeroSense Symposium*, April 1996.

[Mot86] Motorola, Inc. *DSP56000 Digital Signal Processor User's Manual*. Phoenix, AZ, 1986.

[Mot94] Motorola, Inc. *MC 68356 Signal Processing Communication Engine (SPCE) - Product Brief*, 1994.

[OS95] Alan V. Oppenheim and Ronald W. Schafer. *Zeitdiskrete Signalverarbeitung*. Oldenbourg, München, second edition, 1995.

[Phi97] Philips Electronics North America Corp. *Trimedia TM 1000 Preliminary Data Book*. Philips Electronics North America Corp., 1997.

[Pos98] Uwe Post. Winchip 2: neuer Prozessor von IDT. *c't*, (23):30, December 1998.

[RSV97] A.K. Riemens, R.J. Schutten, and K.A. Vissers. High speed video deinterlacing with a programmable TriMedia VLIW core . In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'97)*, pages 1375–1380, San Diego, CA, September 1997.

[RW97] John Rayfield and Dave Walsh. Practical DSP software development on 'Piccolo', ARM's combined RISC/DSP architecture. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'97)*, San Diego, CA, September 1997.

[SA95] Tom Shanley and Don Anderson. *PCI System Architecture*. Addison-Wesley, New York, 1995.

[Sch95] Georg Schnurer. Fire, Fibre, SSA - Die neuen seriellen Massenspeicherschnittstellen. *c't*, (6):126–134, June 1995.

[Sch96] Manfred Schlett. A New Way of Integrating RISC and DSP. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'96)*, pages 653–657, Boston, MA, October 1996.

[Sch98] Friedhelm Schmidt. *The SCSI Bus & IDE Interface*. Addison-Wesley, 2nd edition, 1998.

[SK94] Jay K. Strosnider and Daniel I. Katcher. Mwave/OS: A Predictable Real-time DSP Operating System. *DSP Applications Magazine*, March 1994.

[SO97] Howard Shi and Robert E. Owen. Evaluating DSP Instruction Set Extensions in High-Performance Processors, 1997. Workshop Material, presented at DSP World Spring Design Conference 1997.

[Spe96] Spectron Microsystems, Inc. *SPOX–KNL Performance and Size*, 1996.

[Spe97] Spectron Microsystems, Inc. SPOX Software Developers Kit Available from Spectron, March 1997. Press Release, available at `http://www.dialogic.com/company/presroom/pressrel/sdkrel.htm`.

[SPL92] Lawrence C. Stewart, Andrew C. Payne, and Thomas M. Levergood. Are DSP Chips Obsolete ? In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'92)*, pages 178–187, Boston, MA, 1992.

[ST97] Mark Siggins and Johan Thie. Comparing and Evaluating 'C6x and 'C4x DSPs. Technical Report Revision 1.1, Traquair Data Systems, December 1997. available at `http://www.traquair.com/?c4x.vs.c6x`.

[Sti98] Andreas Stiller. Mandelbrötchen backen. Programmieren mit 3DNow! und MMX. *c't*, (15):186–190, July 1998.

[Sti99] Andreas Stiller. Cindy und Bert. Architekturen von Katmai (Pentium III) und Sharptooth (K6-III). *c't*, (5):124–129, 1999.

[Stu97] Kai Stuhlemmer. Audio-MPEG-Decoder für DSP. Großer Beleg, Dresden University of Technology, July 1997. (in German).

[Stu98] Kai Stuhlemmer. Echtzeitfähiger MPEG-Audio-Encoder für DSP. Master's thesis, Dresden University of Technology, June 1998. (in German).

[Sun95] Sun Microsystems, Inc. The Visual Instruction Set (VIS): On-Chip Support for New-Media Processing. Technical White Paper, available at `http://www.sun.com`, 1995.

[Tec96] TechOnline. Texas Instruments' 'C80 and Machine Vision, August 1996. White Paper, available at `http://www.techonline.com`.

[Tex95] Texas Instruments, Inc. *TMS320C80 (MVP) Multitasking Executive User's Guide*, March 1995.

[Tex96a] Texas Instruments, Inc. *TMS320C4x General Purpose Applications User's Guide*. Houston, TX, 1996. no. D414001–9761, revision *.

[Tex96b] Texas Instruments, Inc. *TMS320C4x User's Guide*. Houston, TX, 1996. no. 2564090–9761, revision C.

[Tex97] Texas Instruments, Inc. *TMS320C80 Digital Signal Processor Data Sheet*, March 1997. available from `http://www-s.ti.com/sc/psheets/sprs023b/sprs023b.ps`.

[TH99] Shreekant (Ticky) Thakkar and Tom Huff. The Internet Streaming SIMD Extensions. Technical report, Intel, Inc., May 1999.

[USB96] *Universal Serial Bus Specification*, January 1996. Revision 1.0.

[Val95] Andrew Valencia. An Overview of the VSTa Microkernel. available from `http://www.zendo.com/vsta/vsta_intro.ps`, 1995.

[Wal96] Dave Walsh. The ARM Piccolo Architecture for Signal Processing – An Innovative New Architecture for Unified DSP and Microcontroller Processing. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'96)*, pages 658–663, Boston, MA, 1996.

[WB97] Ole Wolf and Jeff Bier. DSP Benchmarks – Latest Findings. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'97)*, San Diego, CA, October 1997.

[Wei97] Markus Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures. In *Proceedings of RAW97*, Geneva, April 1997.

[WH95] Jörg Wittenberger and Hermann Härtig. Integration flexibler Hardware. Technical report, Dresden University of Technology, 1995. (in German).

[WK96] Christoph Welte and Cornelius Kellerhoff. Butler – A Real-Time Operating System Optimized for Parallel DSP Applications. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'96)*, pages 1428–1432, Boston, MA, October 1996.

[Yiu98] Henry Yiu. Using the TMS320C80 Multitasking Executive. Technical Report SPRA175, Texas Instruments, March 1998. Application Report.

[ŽMSM94] V. Živojnović, J. Martínez, C. Schläger, and H. Meyr. DSPstone: A DSP-oriented benchmarking methodology. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'94)*, Dallas, TX, October 1994.

[ŽSWS95] V. Živojnović, H. Schraut, M. Willems, and R. Schoenen. DSPs, GPPs, and Multimedia Applications – An Evaluation Using DSPstone. In *Proceedings of the International Conference On Signal Processing Applications & Technology (ICSPAT'95)*, Boston, MA, October 1995.

Index